(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 553 098 B1

(12) EUROPEAN PATENT SPECIFICATION

(45) Date of publication and mention of the grant of the patent:
17.12.1997 Bulletin 1997/51

(21) Application number: **91914430.3**

(22) Date of filing: **12.07.1991**

(51) Int Cl.[6]: **G06F 17/22**, H04L 29/06

(86) International application number:
PCT/US91/04946

(87) International publication number:
WO 92/01251 (23.01.1992 Gazette 1992/03)

(54) EDI TRANSLATION SYSTEM

EDI-ÜBERSETZUNGSSYSTEM

SYSTEME DE TRADUCTION EDE

(84) Designated Contracting States:
AT BE CH DE DK ES FR GB IT LI LU NL SE

(30) Priority: **13.07.1990 US 552080**

(43) Date of publication of application:
04.08.1993 Bulletin 1993/31

(73) Proprietor: **PREMENOS CORPORATION**
Concord, CA 94520 (US)

(72) Inventors:
- **PASETES, Emmanuel, K., Jr.**
  Danville, CA 94506 (US)
- **JENKINS, Lew**
  Pleasant Hill, CA 94523 (US)

(74) Representative: **Altenburg, Udo, Dipl.-Phys. et al**
Patent- und Rechtsanwälte,
Bardehle . Pagenberg . Dost . Altenburg .
Frohwitter . Geissler & Partner,
Galileiplatz 1
81679 München (DE)

(56) References cited:

FR-A- 2 371 098	**US-A- 4 667 290**
US-A- 4 729 096	**US-A- 4 787 035**
US-A- 4 860 203	**US-A- 4 951 196**

- **COMMUNICATIONS OF THE ASSOCIATION FOR COMPUTING MACHINERY. vol. 30, no. 5, May 1987, NEW YORK US pages 408 - 414 S. MAMRAK ET AL. 'A Software Architecture for Supporting the Exchange of Electronic Manuscripts'**

EP 0 553 098 B1

Description

BACKGROUND OF THE INVENTION

This invention relates generally to Electronic Data Interchange ("EDI") systems and more particularly to a novel EDI translation system.

EDI can be defined as the paperless, computer application to computer application, inter- and intra-organizational exchange of business documents, such as purchase orders and invoices, in a structured, application-processable form. An EDI document can be sent directly to a business partner's computer over a communication line. EDI provides many benefits, including: a) speed — documents are sent and received almost immediately; b) accuracy — documents are received as they were transmitted, eliminating manual rekeying of data and attendant errors; c) cost reduction — rapid document turnaround allows more accurate planning of inventory levels and reduces inventory reorder time; d) increased productivity — employees are freed from paperwork and available for other tasks; e) simplified broadcast communications to multiple trading partners (such as sending a request for proposal); f) directness of communication — data is routed directly from the person placing an order to the data processing system of the receiving organization; and g) data integration — data in documents can be integrated directly with existing business information and data processing systems.

EDI involves three essential components: a) EDI standards; b) communications means; and c) an EDI translation system. EDI standards can be divided into formatting standards, dictionary standards, and communications enveloping standards. Formatting standards govern: a) what documents can be communicated; b) what information is to be included; and c) how the information is to be sequenced and presented.

Dictionary standards specify the meaning of the various elements being combined by the formatting standard. Communications enveloping standards define how to group documents together into larger units. Communications enveloping saves on addressing by grouping a number of messages meant for the same destination from a specific source. The communications enveloping standard can also provide password security not present in paper forms of communication. Any of the standards can be proprietary standards (limited to one organization and its trading partners) or common EDI standards (adopted by industry-wide or cross-industry users).

EDI standard documents from and to which trading partner data is converted have detailed definitions in the pertinent EDI standard. EDI standards are maintained by various standards maintenance organizations. Examples of such standards (and maintenance organizations) are the ANSI X12 standard (developed by the American National Standards Institute's Accredited Standards Committee's X12 group), the UN-EDIFACT standard (Electronic Data Interchange for Administration, Commerce, and Transport, an international standard based on ANSI X12 and the Trade Data Interchange standards used in Europe), the Uniform Communications Standards ("UCS"), and TDCC (developed by the Transportation Data Coordinating Committee). The standards are not invariant — they continue to evolve to meet the changing needs of information transfer.

Standards may have different terminology. There are however, similarities in the meaning associated with the terms. Whether termed a transaction set (ANSI X12), a standard message (EDIFACT), or a document (UCS), there is an electronic representation of a paper business document. A unique identifier code is assigned in the standard for each type of business document. As an example, in the ANSI X12 standard an invoice is referred to as X12 document number X12.2, with a transaction set identification code of 810.

EDI standards are developed using a set of abstractions: a) there is a unit of data called a data element; b) data elements can be grouped into compound data elements; c) data elements and/or compound data elements are grouped into data segments; d) data segments can be grouped into loops; and e) loops and/or data segments are grouped into a business document.

The abstraction is based on an analogy to a paper document. Paper documents can be considered to have three distinct areas: the heading area, the detail area, and the summary area. In many cases the detail area consists of repeating groups of data elements. For example, in an invoice, the elements are the items being invoiced and are usually printed as lines in a columnar list. In the terminology used in the standards, these repeating groups are **loops.** Grouping data elements into loops proves unwieldy because of the number of data elements that must be considered. The standards therefore group data elements into data segments and compound data elements.

Transaction set standards specify whether data segments are mandatory, optional, or conditional and indicate whether, how many times, and in what order a particular data segment can be repeated. The transaction set standard does not specify the content of individual data segments. Instead, a segment directory identifies the specific data elements to be included in each data segment. The segment directory is composed of a series of data segment diagrams, each of which identifies the data elements to be included in a data segment, the sequence of the elements, whether each element is mandatory, optional, or conditional, and the form of each element in terms of the number of characters and whether the characters are numeric or alphabetic.

Data segment diagrams include the following components. The **data segment identifier** identifies the data segment being specified. The **data element separator** is a user-selected character that precedes each constituent data element and serves as a position marker. The **data segment terminator** is a user-selected char-

acter used to signify the end of the data element. **Element diagrams** describe individual data elements.

Depending on the standard, element diagrams can define an element's name, a reference designator, a data dictionary reference number specifying the location in a data dictionary where information on the data element can be found, a requirement designator (either mandatory, optional, or conditional), a type (such as numeric, decimal, or alphanumeric), and a length (minimum and maximum number of characters). A data element dictionary gives the content and meaning for each data element.

EDI standard documents are electronically packaged or "enveloped" for transmittal between trading partners. Enveloping can be at several levels. The first, or innermost, level of enveloping separates one document from another. This is accomplished by attaching transaction set headers and transaction set trailers to each transaction set, or document. At a second level of enveloping, documents can be packaged together into groups known as functional groups. An example of a functional group is a purchase order and an invoice, which are often sent together in both the paper and EDI worlds. Each functional group is packaged with a functional group header at its beginning and a functional group trailer at its end. This second level of enveloping is an optional level in most standards.

At a third level of enveloping, all functional groups to be sent to a single trading partner can be packaged together. This enveloping consists of an interchange control header and an interchange control trailer bounding either the packaged functional groups and/or the document.

The second component of EDI is communications means. EDI standard documents are transmitted electronically between trading partners' computers. The transmittal can be directly between trading partners, via a direct private network in which the computers are linked directly. Direct networks become difficult to maintain with larger numbers of trading partners. The alternative is to use a third-party, or value-added network (VAN). A VAN maintains an electronic mailbox for each trading partner that can be accessed by each other partner (with appropriate security restrictions).

The standard does not specify a communications standard except to the extent to which it describes the enveloping standard and the way in which transmissions can be acknowledged. There are de-facto standards such as IBM 2780/3780 BSC protocols used as file transfer protocols. Virtually every EDI VAN supports these protocols because of their pervasive use.

The third component of EDI is the EDI translation system, which performs at least the functions of data communication and document translation. Document translation is the most significant function in this component, and is implemented through an EDI translation software.

In practice, participants in EDI select and agree to use a proper subset of the document standard. Such agreements among trading partners are too numerous to be included in an EDI translation software. The EDI translation therefore needs to provide a way for the user to express how to process or generate the EDI document in compliance with the agreement.

One example of the need for compliance with a trading partner agreement is when one trading partner is to automatically post an EDI document to a processing system. A purchase order received from a customer could be booked automatically through the recipient's order entry system. Since the agreement and the interface requirements to the order entry system are not defined in the EDI translation software, the user of the EDI translation software must specify how to perform the translation.

The EDI translation software must provide three functions in conjunction with giving the user a means of expression for performing the translation. First, the software must provide mechanisms to navigate and manipulate EDI documents. Second, the software must be able to produce the records that are to be interfaced with the order entry system. Third, the system must provide the user with means to express (using these primitives) a complete procedure for transforming an EDI document into something that is compatible with the user's interfacing requirements.

In some cases a user may find it advantageous to automatically print an EDI document on paper in a format that is familiar to the user. For example, it may be helpful to print an invoice because the accounts payable system is implemented manually. As in the discussion above, the translation software does not include a definition of the format in which the user expects the output. The user must therefore specify this information.

The development of prior EDI formatting or translation software can be traced through three generations — translation software of the invention represents the fourth generation. The first generation of software was developed in the late 1970s and early 1980s to support a variety of private formats. The private formats were developed by large corporations to allow them to exchange business documents with their trading partners. The formats did not conform to any industry standards. The software used in these systems resembled subsystems of existing general business applications.

These first generation systems typically involved the exchange of private-format data files and associated data processing programs. Trading partners typically communicated directly with each other over private networks. Examples of first generation systems include General Motors' and Ford's automotive "release" systems and retail industry order processing systems, such as those used by Sears, J.C. Penney, and K Mart.

The document FR-A-2 371 098 discloses a system for transmitting numeric data, employing a commutation of messages in terms of packages. A sort of de-envelopping procedure (in view of communication of data with

different format) is present in the operation of this system. The document teaches that there is an increasing need for transmitter-receivers in the terminals of existing communication networks requiring commutation of numeric data. It is essential that the terminals have the capability to communicate with each other despite the diversities of codes, modulation techniques and protocols. The document also teaches that communication processors have been employed in the past for the conversion of codes and other data. The known system is able to convert the data received from various sources into a format having a protocol adapted to the system. In particular, the known system may perform facsimile transmissions having a commutation of packages and having a protocol which is adapted to the ensemble of the system.

The second generation was characterized by the introduction of variable length, hierarchical document standards such as TDCC (used in the transportation industry) and UCS (used in the grocery industry), which created a need for a more generalized approach to translating those standards into computer-processable business forms.

Translators of this generation typically employed fixed data files transferred to an intermediate process that translated the document into a form usable by host applications. The translator's primary task was to convert the records in the data files from variable length format to a fixed length format that could be processed by traditional batch applications. These translators were known colloquially as "asterisk strippers" because they had no capability to manipulate data between records or to change the placement of data within records.

A major shortcoming of these second generation translators was that a large amount of additional intermediate processing was required, typically involving programming tasks to integrate the EDI document into an application system. Networks and major trading partners tended to overlook this problem in their efforts to expand trading partner relationships. The dissatisfaction of end users with the high software maintenance demands of this post-translation processing requirement led to the development of the third generation of software.

The document Communications of the ACM, Vol. 30, No. 5, May 1987, New York, US, S.A. Mamrak et al.: "A software architecture for supporting the exchange of electronic manuscripts", pp. 408 - 414 discloses an electronic manuscript exchange tackling the known problems in translating among the wide variety of electronic representations. The system supports both the use and the creation of translation tools.

The third, or current, generation of translation software attempts to provide higher levels of transform capability so that an EDI document can be put into a form closer to the input required by the user's integrated application software. These translators provide for table-driven systems and "dynamic mapping."

The translation software uses a table structure to perform the translation. The tables consist of the standard data dictionary and syntax rules for the data segments and elements of a given EDI transaction set. The software selects the appropriate table to perform the translation for a specified EDI transaction set to be generated.

Dynamic mapping allows the user to identify the relationship of elements within a segment to fields in an application input document and vice versa. Instead of fixing record lengths, the systems allow the user to put data elements into different data files in any location. Rather than being limited to a single fixed length file in a transaction set, the user can select data from multiple files, in any order within the file, and present the data to the translator.

The ACS Network Systems EDI 4XX product, available from ACS Network Systems of Concord, California, is typical of third generation software products. The data communication component of ACS 4XX provides the means to generate and maintain a communication line directly with a trading partner or to a third party data network and the means to control the process of sending and receiving documents to and from trading partners. The translation component of ACS 4XX translates incoming standard business documents from an EDI Standard format to a format usable by applications programs and reverses the process for outgoing data.

These third generation systems suffer from several problems. First, the dynamic links between fields in application databases and data elements in EDI documents are unconditional — the field mapping or linking is construed to be constant in any given application. The systems are therefore unable to represent the conditional expressions that appear as "notes" in almost all standards. For example, a segment definition may have a conditional note that specifies that either the second data element must be present or both the third and fourth elements must be present. Although some translators have attempted to comply with these notes through the actual translator software code, this technique has proven inadequate and difficult to maintain as standards and documents evolve. Further, the standards definitions employed by these systems have relied on a data base schema of the standards definition This is a relatively inflexible approach that does not readily accomodate the continual evolution of the standards.

Second, these systems assume that EDI input will require non-EDI, or application, output. This prevents the systems from acting as true translators, capable of communicating any type of input and any type of output. Similarly the communication components of such systems assume that their only interfaces would be with EDI-capable networks. This precludes straight file transfer capability between computers, and prevents the systems from acting in a terminal emulation mode to interface with other computers. Further, the communications interface does not provide the structure for unattended

operation; the systems cannot act as passive communications systems for receiving calls from outside systems. There is also no method for allowing an outside application to send the system data for translation into EDI format.

Third, there can be structure or sequence clash between the EDI document and the application transaction's structure requirement. The previous generation systems provide a tool for specifying transformation of data through the use of a mapping program. The mapping program includes assumptions about the correspondence of structure and sequence between source data and target data. The tool therefore cannot be used to translate data with a structure or sequence clash between the source data and target data.

EDI documents are defined using a specification that prescribes a sequence and structure to the document. When the application transaction's structure and sequence differs from the corresponding EDI document specification, the user is required to develop a set of programs that deal with the structural or sequence difference to achieve a seamless interface to the application.

For example, a retail store may send to a manufacturer purchase orders with store distribution specifications attributed to each line item. If the manufacturer requires separate paperwork for each shipping location, the grouping of data as expressed by the EDI document clashes with the manufacturer interface requirement. The clash is caused by the different structure assumed when producing the EDI document (which specifies how to distribute the order for a specific item to multiple shipping locations) and the structure assumed by the order entry system (which assumes that an order document contains items for a specific shipping location).

Third generation systems do not attempt to address this problem. The general approach is to produce a file that contains the necessary data in a rigid, hierarchical structure and to force the user to develop a set of programs using whatever programming language and utilities are available to change the structure. Some of these programs can become very complicated.

Fourth, third generation translators generally have limited pattern matching capability. This reduces the scope of acceptable types of input. The pattern matching of application transaction files is very limited. Records are identified using a strictly specified value in a specific position in the record. Therefore, unless the application transaction file created from the computer application contains these very specific value(s) (i.e, H01, H02, D01, D02, etc.), an interface program must be developed to supply data to the translator.

Fifth, the current systems have strict one-to-one correspondence between source and target documents. The systems therefore cannot receive a document, such as a shipping notice, and generate multiple transactions, such as a receiving notice, an inspection notice, and an invoice notice.

Sixth, prior systems have limited capacity for evaluating performance errors. For example, a user who creates a mapping specification with a table driven or dynamic mapping scheme may find during operation that some of the data elements are output incorrectly — the value of a data element for one document may turn up in another document. It can be difficult for the user to find the source of the error because the systems provide no debugging mechanism for finding the problem. While a debugger capability could theoretically be added, it would be onerous to implement because the structure of the systems (being non-language based) is not susceptible to a debugging facility.

Finally, the third generation translators limit operations to simple assignment semantics. There is no provision for performing arithmetic operations on source data elements to produce a target data element. Performing logical operations or string manipulation are usually not provided.

SUMMARY OF THE INVENTION

The invention, as claimed, overcomes the problems with the prior generation systems with an EDI translation method that receives data from a source in one format, executes a script to translate the data into a second format, and transmits the data in another format to a destination. The system has the ability to transform input data from, and into, virtually any format and to produce more than one output for each input document. The system employs a tree data structure and a script of translation instructions to overcome the hierarchical data structure limitations imposed by the prior generation systems.

The system can pattern-recognize records that are not explicitly differentiated. It provides flexibility in using virtually any communication system to communicate EDI as well as non-EDI documents. The system employs a model that assumes that both input and output are to be communicated and is therefore not constrained to use with a particular processor, but can instead act as a true communication front end.

The system addresses the difficulty that prior systems had in expressing syntax notes associated with segments in relational terms. This is done by expressing the notes using language construction of logical operators on data elements. For example, the semantic expression "(-or 2 (-and 3 4))" means "either the second element or both the third and fourth elements must be present."

The system addresses the inability of prior systems to handle structure clash between an EDI document and an application's data structure requirement by providing data and control structures. The control structures include such basic structures as executing a series of commands while some predicate condition exists.

The system supports data structures such as multidimensional arrays and an EDI tree data structure. The EDI tree data structure represents an instance of an EDI

document. The tree provides random access operation on the document allowing the user to specify the sequence in which access to the EDI document is to be made. The system further provides a set of access primitives. These two tools allow the user, for example, to read a document and process it repeatedly, once for each of several shipping locations.

The system facilitates support for relational level notation in EDI document definitions that specify hierarchical nesting, such as EDIFACT standard documents. It supports implicit as well as explicit notation in EDI document definitions. The tree access notation enables the EDI programmer to reference EDI data elements so that a language can support it as one of its data types. Without such a notation, support for EDI in a programming language would be difficult. Nesting and repetition of segments is discussed in ISO Publication 9735, pp. 6-9 (1988).

The system also overcomes the limited ability of prior systems to pattern match data by providing a pattern matching capability that can read a fairly large set of patterns that follow an LL(1) grammar (for a description of LL(1) grammar, see Aho, Sethi, & Ullman; <u>Compilers: Principle, Techniques and Tools,</u> pp. § 4.4 (Addison Westley 1986). The user can formulate a filemap definition and record definitions to express the expected pattern. The system also allows the user to generate as many target documents as required from a single source document.

The system also overcomes the failure of the prior systems to provide a debugging capability. Since the system is language based, a conventional debugging facility can be readily provided.

Finally, the system has a broader range of operations performed than the prior systems and provides expressions such as those involving logical and arithmetic operators and string manipulation. Data transformation is enabled by use of an assignment statement. The assignment statement accepts an expression on the right hand side as in:

$$a = b + c$$

This statement specifies that a is computed from b plus c. The elements a, b, and c could be elements from the source or target, although a is likely to be an element of the target and b and c are likely to be elements of the source. In this example, a "+" operation is used in the right hand side expression. The system supports many such operations including "substring" to extract a portion of a string. These operations are termed language "primitives". The set of language primitives depends on the nature of the problem. However, by employing a programming language implementation, the invention allows a user to combine these basic operations into complex expressions. This expressiveness allows the user to specify more complex transformations.

The system is organized into four component work centers:

a) Communications Interface (having a communication session as its input work unit); b) De-enveloping (having an interchange as its input work unit); c) Translation (having a document as its input work unit); and d) Enveloping (having an enveloping request as its input work unit).

The communications interface work center uses a script to schedule a communication session and describe how to break up the contents of the communication into units of de-enveloping work.

The de-enveloping work center divides a communication interchange into its component documents. It also performs a routing function, routing documents to the required destination.

The translation work center manipulates an incoming document into the format that is expected by another system. It can convert EDI data to a format that can be printed or used by application programs and can convert a file created by an application program to a standard EDI format. It is implemented as an interpreter or compiler that understands translation primitives and can be used with a script to perform transformations on many kinds of data. In the illustrated embodiment, each of these work centers are implemented using a novel EDI programming language, referred to herein as the e-language.

<u>BRIEF DESCRIPTION OF THE DRAWINGS</u>

FIG. 1 is a block diagram of data flow through the EDI translation system.

FIG. 2 is a block diagram of a hardware implementation of the invention.

FIG. 3 is a sample communication script.

FIG. 4 is a sample EDI de-enveloping script.

FIGS. 5A to 5F are a sample application de-enveloping script.

FIG. 6 is a graphic representation of the organizational structure in which files are stored before enveloping.

FIG. 7 shows sample application data input for a translation operation.

FIG. 8 shows sample EDI output corresponding to the input of FIG. 7 as produced by the script shown in FIG. 13.

FIG. 9 is a document definition for an X12 810 invoice implemented in the e-language.

FIG. 10 shows two sample segment definitions implemented in the e-language.

FIG. 11 shows two sample element definitions implemented in the e-language.

FIGS. 12A to 12C show three sample data type definitions implemented in the e-language.

FIGS. 13A to 13F show a sample translation script for translating an application document to an EDI document.

FIGS. 14A to 14F show a sample translation script for translating an EDI document to an application document.

DETAILED DESCRIPTION

The system is organized into four component work centers: a) Communications Interface; b) De-enveloping; c) Translation; and d) Enveloping. The flow of data through the system is illustrated in FIG. 1. Data flows into the system through communications interface 1. It then flows to the de-enveloper 2, to the translator 3, and to the enveloper 4. From the enveloper, the data flows back to the communications interface and thence out of the system.

In the illustrated embodiment, a data item flowing through the system can be considered conceptually as a package with a packing slip. As the data item is processed by each component of the system, some information is read from the packing slip, and additional information is written to the slip. The information that appears on the slip determines how the package will be processed. The system uses a "routing form" as an abstract representation of the packing slip. A routing form follows a data item through the system, and a component can both read from and write to the routing form. The behavior of a component may depend upon information that is read from the routing form.

The routing form provides for the following information: a) interchange sender; b) interchange receiver; c) functional group sender; d) functional group receiver; e) node (the network or application used); f) facility (the communication protocol); g) content type (EDI, application, or text); h) document type (transaction set identifier); i) translation script; j) functional group; k) error message; and l) functional acknowledgement flag. The information is placed in the routing form in the following order: a) in the communications interface - the node, facility, and content type; and b) in the de-enveloper - document type, senders, receivers, functional group, functional acknowledgement flag, and translation script. Error messages are written to the routing form by whichever module detects the error.

The EDI translation system can operate on a variety of hardware. In the illustrated embodiment, the system operates on a microcomputer having a 20 MHz Intel 80386 processor, 4-8 MB of RAM, one or more serial ports, a 100 MB fixed disk drive, a 2400 Baud modem with V.22 VIS/MSP protocol, and having a UNIX operating system. The system may also be operated on a local area network, with different machines implementing different functions of the system. For example, one machine can serve as the communications interface while another provides file storage.

As shown in FIG. 2, the system operates on microcomputer 10, which is connected to one or more EDI networks 20 and a host 30. The connection 40 between the system and the host is a synchronous connection such as a Bell 208A modem or another line driver. EDI data is transmitted to and from trading partners via the EDI networks, while application data is transmitted to and from applications operating on the host. All data transmission between the system and either the host or the EDI networks is routed through the communications interface work center, shown at 50.

In an alternate embodiment, the host on which the applications are operated and the microcomputer on which the EDI system operates are combined, so that applications are operated on the same machine as the system.

In the illustrated embodiment, each of the EDI system work centers is implemented in the e-language. Of course, the work centers could be implemented in other languages, such as Lisp or C+ +, although with greater difficulty. The e-language uses arithmetic, boolean, functional, and control structures similar to those in BASIC, Pascal, and COBOL. In addition, the e-language contains functions and data structures unique to EDI processing. These specialized functions include facilities for converting an EDI document to an EDI data tree and for performing the reverse operation. The e-language also contains commands to determine the status of various parts of the EDI system, and to perform low-level operations on these parts. These functions, and other aspects of the e-language, are described in the "e" Programmer's Reference Manual, attached as an Appendix hereto and being a part of the disclosure of this application.

The e-language allows the user to perform pattern-matching operations. For example, the e-language can be used to describe the format of an application file and to select the portions of the file to be used. This allows the production of a customized EDI document from an application file.

The e-language is structured in the manner described in the attached Appendix. The syntax of the e-language borrows some of the command names and clauses from BASIC and COBOL and general syntax structures from Pascal.

The system employs the concept of a data stream. A stream is a source of input or a destination of output. An input/output function generally takes a stream as one of its arguments. When a program is executed, a default input stream and a default output stream are created. Stream objects are bound to one or the other stream. Other streams can be opened during a program execution that can be bound to variables used in other runtime calls for input/output purposes.

Communications Interface

The communications interface has two parts: a) a scheduler; and b) a set of facilities drivers. The scheduler is a program that references a usercreated table to determine when to communicate with sources of information (such as a trading partner or an application). The

actual communication is done through a facility driver. All of the mechanics of transferring files via a given communication protocol are packaged in the facility driver, and are accessible to the user through a communication script. The script, which can be written in the e-language, specifies the procedures to be used to create a de-enveloping work unit.

The communications interface work center defines the concept of a logical connection to the outside world as a node. The logical implementation of the connection is a communications facility. Two kinds of nodes are defined: a) an EDI connection (or EDI network); and a non-EDI connection (or application). Thus, in FIG. 2, EDI network 20 and an application running on host 30 are each nodes. The node name is used to symbolize the network or application, binding the name to the specific communication facility that is used to execute the communication. In the illustrated embodiment, the system assumes that communication facilities exist in the form of hardware and software. These facilities are not part of the system; the system interfaces to the facilities through facility drivers, which are part of the system. The interface consists of a **driver** and a **script.** A driver is a program capable of communicating with the specific communication facility. This driver supports a set of primitives that is supported by the e-language. Communications packages usually have either a programming interface or a user interface. The driver must be interfaced to whichever interface the specific communication package has. A script is a program capable of configuring the communication package. This gives the system the ability to set up and use the proper script.

The different facilities to which the system could interface are numerous. It is therefore economical to have interfaces for common facilities such as Unix Mail, Unix File System, and Remote Job Entry. The operations provided by the interface depend on the capabilities of the facility, but as a minimum include send and receive primitives.

A facility client is capable of both sending and receiving files. The EDI translation system can be connected to a network node using a Binary Synchronous Remote Job Entry facility. The facility may be implemented by software such as the AT&T Synchronous Gateway. The gateway uses an account name to manage communication. When establishing communication with the network node in conjunction with the gateway, the network node is mapped to an account name, thus integrating the gateway communication software with the EDI system. A similar correspondence can be established between an application node and the gateway account. Other facility clients can include Unix Mail, X-MODEM, and the file system in the operating system.

Each node can also be assigned a schedule specifying the times when activity with the node is expected to occur. The schedules are derived from configuration specifications that users provide. The configuration specifications are configuration texts, rather than e-language scripts. Each schedule specifies the time and an associated script that is expected to be executed. The script, which, in the illustrated embodiment, is implemented in the e-language, contains details of how the communication is to be performed. A sample communication script for a Unix Mail interface is shown in FIG. 3. The script embodies four primitives. The first, shown at 8110, is **Envelope,** which request that envelopes be made for data that is ready to be transmitted through the Unix Mail. The ATTMAIL parameter, 8120, identifies the network. The enveloped data is returned as a data stream. The second primitive is **SendUnixMail,** shown at 8210, which sends the enveloped mail messages in the stream out through the Unix Mail facility. Next, **ReadUnixMail**, shown at 8310, picks up from ATTMAIL any mail message, and develops a list of mail messages. Finally, **MakeMail Interchange**, shown at 8410, converts a mail message into an interchange.

This example illustrates the use of interface primitives that were developed as part of a Unix Mail driver. The example uses the Unix Mail facility to send and receive messages through the ATTMAIL network. If the pertinent parameter (such as 8120) were changed, the script could equally well use any other network. Scripts for simply reading mail without sending mail, or sending mail without reading mail, could also be developed and scheduled to execute at different times.

The scheduler is driven by a scheduler table, in which the rows are nodes and columns are facilities. Each cell in the table can contain a schedule instruction. The schedule instruction can specify that a communication should occur at a fixed interval or at a fixed time.

A facility is an interface with specific communication resources provided in the host machine. The facility includes information about how to use the communication resource. Because each facility is programmable, and therefore variable, a command must be issued to specify the communication script that the facility client should execute. The script is specified in the scheduler table.

Data received from a communication session is sent to the de-enveloper as a unit of work. The unit of work is assigned a script name, which identifies the de-enveloping procedure that will be used to divide the contents of the communication session.

The system also provides the structure for unattended operation, acting as a passive communications system for outside systems to call and to send it data for translation into EDI. The structure, based on the driver and configuration described above, allows the system to take advantage of existing communications packages. One of the standard facilities provided with the system is a file system communication facility. The facility consists of a listener that polls specified directories for files and a depositor that places enveloped data into a target file or separate files into a target directory. Files with specific names or files found in a specific directory are picked up by the system. The file name or the source directory determines the type of data and the require-

ments for de-enveloping.

The user can therefore use the file system communications facility in conjunction with any communication system that places incoming data into a specific directory to simulate incoming support. The user can also develop a program to transmit data deposited in a particular directory or file to the communication program.

These capabilities provide the tools for running unattended for communications initiated by another party. The system provides a scheduling mechanism for each network interface and application interface to allow the automatic scheduling of communications initiated by the system. The scheduling is done using a facility like timer services provided by some operating systems, such as the "cron" facility of the Unix operating system, which provides for the execution of specified commands according to a time schedule.

De-Enveloping Work Center

When incoming data contains multiple documents in one unit of work, the de-enveloping work center of the EDI system must break the work into single documents. The de-enveloper has three purposes: a) to strip off the enveloping layer and divide a communication interchange into its component documents; b) to identify the sender, receiver, and transaction type; and c) based on that information, determine the type of transformation needed and the destination of the transformed document, saving the enveloping information as attributes of each document. The process is script driven. The content type and the de-enveloper script are determined by the communications process; the identity of the interface is known at the time the communications process located the data. If the data was located by a network interface, the content type is EDI and the EDI de-enveloper script is invoked for that network interface. If data from an application interface was located, the content type is application transaction and the application transaction de-enveloper script is invoked.

Script selection is performed through a table, the table having nodes as rows and the type of data as columns (being either EDI data, applications data, or text). In the case of EDI data, the de-enveloper work unit (an interchange) is de-enveloped into its component functional groups and then into transaction sets. Each individual transaction set is then considered a separate translation work unit. In the case of an application file, the report is divided into individual invoices. Each individual invoice is then considered a separate translation work unit. The system has the capability to handle text files, but performs no de-enveloping, translation, or enveloping on them.

The de-enveloping process is also script driven. Different scripts, and methods, are used depending on whether the data is EDI or non-EDI. The system gives the user the capability of defining each script to meet the user's particular requirements.

For an EDI input, the de-enveloping script uses the following general procedure. First, the start of the file is marked. Then, for each found interchange, the following steps are taken. First, data from the last mark to a position just prior to the interchange envelope is returned as a de-enveloper unit of work, considered an application transaction with the same name as the network interface. Second, the interchange envelope is read and an attribute list consisting of interchange values is populated. Third, for each functional group envelope found within the interchange envelope, the following steps are taken. First, the functional envelope is read and additional attributes are populated. Second, for each document envelope found within the functional group envelope, the following steps are taken. First, the document envelope is read and document attributes are populated. Second, a mark is set. Third, a translation unit of work is created using data from the mark to the end of the document. Fourth, additional document attributes are populated from the end of the document. When no additional documents envelopes are found, additional functional group attributes are populated from the end of the functional group. When no additional functional groups are found, a mark is set. The entire process above is repeated until no more interchange envelopes are found.

A sample de-enveloping script for EDI data is shown in FIG. 4. The primitive OpenInterchange, shown at 9120, specifies the types of interchange envelopes expected to be present in the data (ISA interchange envelope types in the example). The variable "cursor," shown at 9110, is used to point to the current position in the input/output stream. The primitive "Deenvelope," shown at 9210, extracts a document from a found envelope. The example script extracts documents from all envelopes that it finds.

The de-enveloper reads just enough of the interchange file to extract a transaction and then sends the data obtained to another work center for further routing through the system. In general, data is passed from one work center to another using queues. A configuration file is used by all the work centers to determine what work centers are available to the system.

The de-enveloper tracks the interchange and functional group information as it processes the interchange. Users can access the information through the element reference notation, as described below.

In the case of an application transaction, the de-enveloping process is different. The user must specify how the data is to be divided into individual transactions because the patterns that specify the boundaries between documents are not known. The user employs the pattern matching capability of a function termed SPLIT to split off the individual transactions. The pattern matcher is implemented in the e-language using a technique similar to the recursive descent LL(1) parsing technique, with backtracking to support optional constructions. The pattern matcher uses a specification termed a filemap

to describe the structure of a file and a record specification to describe the structure of each type of record found in the filemap.

The filemap describes the order of records found in the incoming file. The record definition describes the structure of each record type. The language borrows from COBOL picture clauses for specifying the data type of each data element. The structure of the mapfile is shown below:

1. L_MAPFILE - primitive for 'e' mapfile intrinsic

    a. MATCH - high level match given a filemap

        i. MATCHPRIMITIVE - start of the primitive

            (1) MATCHRECORD - match the next record as specified

                (a) INITBINDINGS - initialize the connection with data elements
                (b) MATCHSIGNATURE - check if the record has the proper signature

                    (i) PARSESIGNATURE - understand the signature specification
                    (ii) DOMATCHSIGNATURE - apply the specification
                        1) EVALVALEXPR - evaluate the predicates

                (c) MATCHFIELD - match the specification of the next field
                    (i) MATCHVALUEIS - match the user specified field value
                (d) MATCHSTARFIELD - special field matcher (varying length)
                (e) EXECUTEBINDINGS - connect data to symbol

            (2) EVALMATCHACTION - execute any action associated with matching the record
            (3) MATCHFILEMAP - if nested filemaps
                (a) MATCH - recursively match (to allow backtracking)

        ii. MATCHOPTIONAL - if the next construction is optional
            (1) MATCH - recursively match (to allow backtracking)
        iii. MATCHLOOP - if it is a loop
            (1) MATCH - recursively match (to allow backtracking)

Using a filemap specification, a system user can specify both structure and sequence with minimum ambiguity.

The e-language includes three primitives used in dividing application documents from an application communication. The primitive used to split off documents is SPLIT <stream, filemap>, which reads the file and processes it using the filemap. Additional primitives MARK_END, DO_SPLIT and ROUTE_TO_xxx are used to do perform functions similar to those provided in the EDI de-enveloping procedure. These primitives are to be used in the actions associated with records matched by the filemap specification. MARK_END marks a possible end of a document. DO_SPLIT splits out the document identified with the beginning and end marks and places a new begin mark immediately following the previous end mark.

Application documents are separated by selective use of these primitives in the filemap. For example, a communication file may contain a set of invoice reports. These reports are printed a page at a time, but may be printed in multiple pages. The filemap specifies how to pattern recognize each page and to extract from each page the trading partner's identifier. A begin mark is set at the beginning of the first page and an end mark set at the end of each page. If more than one sequential page includes the same invoice identifier, the SPLIT continues to pattern match until it reaches a page for which the invoice identifier is different, then resets the end mark to be at the end of the last page of the group of pages with the same invoice identifier. The DO_SPLIT primitive is then used to divide out the invoice between the begin and end marks. A sample de-enveloping script for application data is shown in FIGS. 5A to 5F.

The script first identifies the filemap specification, "invoice" to be used, as shown at 10110. The "record [name]" commands, as at 10120, indicate that certain commands are to be executed when the named record is found in the input data stream. The "printf' commands shown, as at 10121, print out specified information for diagnostic purposes. As shown at 10300 in FIG. 5C, each time a record "total3" is found (10310), an end mark is set (10320). As shown at 10400, the script then splits the record with the DO_SPLIT command (10410). The script then determines at 10420 the receiver, using the data element "custnum" shown at 10121 in record "terms" at 10120. Then, in line 10430, the script resets the start marker. In the following steps, as at 10440, the format for each record to be read is defined.

At 10520 in FIG. 5F, the script directs that each invoice be split, keeping a list of items in the variable "vfl, " using the filemap "invoice" and the data from the standard input stream "stdin." Finally, at 10600, for each item in the list "vfl", the item is routed to the translation work center (10640).

As noted in the preceding script, the de-enveloping work center also routes documents to the required des-

tination. The routing function is controlled by a routing table. For an EDI document the following document elements determine how to process the document: a) sender, found from the interchange or functional group envelope; b) receiver, from the same source; and c) document type, which is from the document envelope. Sender and receiver may be known by different names from different sources. The system resolves the different aliases into a specific enterprise using an alias table. For an application transaction, the following data elements determine how to process the transaction: a) sender, derived from application interface profiles or from document scanning; b) receiver, from the same source; c) the application transaction, which comes from the document.

The user specifies how to process translation units of work by entering in the routing table the following information: a) script to be used in the translation; b) the destination network or an application transaction; and c) sender and receiver. This information is found in the routing slip for the work unit.

Translation Work Center

The translation work center manipulates an incoming document into the format that is expected by another system. It can convert EDI data to a format that can be printed and/or used by applications and can convert a file created by an application to a standard EDI format. Unlike existing systems, there is complete flexibility in the ability to translate. It can convert from one EDI format to another or from one application format to another. In the illustrated embodiment, it is implemented as an interpreter that understands primitives of the e-language and is used with a script written in the e-language to perform transformations on many kinds of data. Although shown as implemented as an interpreter, implementation through use of a compiler is also possible.

The translation work center is explained in the context of a sample conversion. A sample input document, an invoice, will be converted from the application format shown in FIG. 7 to an EDI format, which is an EDI 810 document, as shown in FIG. 8. A document definition is created in the e-language that describes the structure of the document and a corresponding bare tree is constructed. Data is read from the input document and mapped onto the bare tree. The data is then mapped from the populated tree into the EDI 810 document format.

The EDI system manages an EDI data stream with an EDI reader and an EDI writer. The purpose of the EDI reader is to develop an EDI data tree from an EDI document expression. The e-language provides the capability to make a bare tree corresponding to a particular document and for creating an EDI tree from a virtual file whose body is a document.

Trees are one type of data variable used in the e-language. In turn, data variables are one type of e-language program element. The elements of a program written in the e-language are described below.

Lexical Elements — The e-language recognizes a character set that includes alphabetic characters, numerical characters, and special characters. **Identifiers** are names given to variables, functions, subroutines, patterns (records), and filemaps in the program. **Constants** can be string or numeric constants. **Reserved words** are words with special meanings in the e-language that cannot be used for program names or identifiers.

Variables — A variable is a name used to refer to objects in the system. The object may be a single element or multiple elements. The three variable types available in the e-language are arrays, records, and trees. Arrays are collections of data values whose individual members are accessed via a numeric index. A record is a compound data structure containing multiple values, or fields, of various types.

As described in more detail below, trees are composed of nodes, with an uppermost node called a root node (or root). Branches below a node are called the node's children or branches. The lowermost or outermost nodes (those with no children) are called leaf nodes or leaves.

The e-language provides the capability to make a bare tree appropriate for a particular document and to create an EDI tree from a virtual file whose body is an document. In both cases a pointer to the root of the tree is created; by assigning pointers, any node of the tree can be referenced.

Expressions/Operators — Expressions are constructs having a well-defined value in the context in which they appear. The simplest expressions in the e-language are identifiers, string constants, function calls, array references, and tree (or node) references. Operators are characters that designate mathematical or relational operations.

Statements — Statements are instructions to the computer to perform some sequence of operations. The types of statements available in the e-language include: a) assignment statements; b) conditional statements; c) iterative statements; d) IF statements; e) FOR statements; f) WHILE statements; and g) subroutine calls.

Functions — Functions are collections of instructions that return a value. Functions include multi-valued functions that return more than one value, subroutine definitions, and filemap definitions. Filemaps are function-like objects that perform pattern-matching on files. A filemap accepts as input a list of records and/or other filemaps, and matches items in the current virtual file with the given records or filemaps. Fields in a record that is successfully matched are bound to the items in the input file they match. The program can then access the values of the fields elsewhere.

Trees are information structures particularly well suited to the structure of documents. By representing documents as trees, the contents of the documents can

be accessed in an expressive and convenient form. The tree is like a file buffer, with a definite structure, and in the e-language, a variable can be assigned to the tree.

Every node in an EDI tree can be classified as one of the following types. An **area** consists of a non-empty sequence of segment blocks and loop blocks. It can be considered as the root of an arbitrary subtree. Its subnodes are segment blocks and loop blocks. A **segment block** represents a block of repeated segments with identical segment identifiers. Its subnodes are segment bodies. A **segment body** represents a single segment within a segment block. Its subnodes are the elements of the segment. A **loop block** corresponds to one or more iterations of a loop. An iteration of a loop is the sequence of segment blocks for the segments marked as a loop in a standard document. The subnodes of loop blocks are loop bodies, analogous to segment blocks and segment bodies. A **loop body** is an iteration of a loop; its subnodes are segment blocks. An element has the same meaning for an EDI tree as it has for an EDI standard - it is the smallest logical portion of a segment. Some EDI standards also employs the concept of composite data elements, which have similar semantics and implementation to segments.

Any node in an EDI tree can be referenced by specifying its path. Symbols used in defining EDI documents are used to develop the path specification. Three operators are used: separator, selector, and arrow operators. Separator operators separate nodes in the tree, while selector operators select a loop body, segment body, or element in that node. An arrow operator points to an NTE segment or a subtree of a hierarchical level loop. An NTE segment is designated in the X12 standard as a floating segment, one that can appear anywhere within a document. Hierarchical levels, used in some EDI documents, show the relationships among different levels of detail in a shipment, such as pallets on a truck, part numbers on the pallet, cartons of a particular part number, etc. Such a notation is necessary because an NTE is considered an appendage to any segment body and an HL subtree is considered an appendix of an HL loop.

An EDI tree is constructed based on a document definition. A document definition for an X12 810 invoice transaction set, implemented in the e-language, is shown in FIG. 9. The definition is divided into a heading area 1100, a detail area 1200, and a summary area 1300. The heading area includes segments, such as segment ST, shown at 1101, and loops, such as loop_n1, shown at 1110. The document definition gives the "grammar" of an invoice document as agreed to by the X12 committee.

Segments and elements can also be defined as text files in the e-language. Two sample segment definitions are shown in FIG. 10, expressed in Lisp. These segment definitions are for the CUR segment (shown at 2100) and for the N1 segment (shown at 2200). Each segment definition identifies the name of the segment, such as N1 shown at 2205, and the title, such as "NAME," shown at 2210. It specifies a list of document definitions in which the segment is used, as shown at 2220. This line indicates that the N1 segment is used in the 810 document definition (indicated at 2225).

The segment definition next defines a syntax. The syntax definition is used to evaluate a particular segment expression to determine whether it complies with the standard. This addresses the shortcoming of prior generation translators in inadequately handling the conditional notes in segment definitions. For example, segment N1 may have a conditional note that either the second data element must exist or both the third and fourth data elements must exist. This note is implemented in the syntax shown for segment N1, shown at 2230. The conditional is expressed in the statement "((-or 2 (-and 3 4)))," where the numbers refer to the data elements.

Next, the segment definition specifies a list of elements, as shown at 2240 for the N1 segment. Each element is indicated as being either mandatory (M), optional (O), or conditional (C). This ties into the syntax. For example, since the syntax provided that either the second data element or the fourth and fifth data elements must exist, these data elements (element numbers 93, 66, and 67) are indicated as conditional. Element 98 is mandatory.

The position of an element in a segment is used in the e-language to refer to the data element. In this element reference notation, the path specified by "tree\HEADING\LOOP_ N1[3]\N1[2]" refers to the second element in the N1 segment, which means that element number 93 is being referenced.

Finally, the segment definition specifies the type and version of the EDI standard being used. As shown at 2250, the N1 segment definition is based on version 0.0 of the X12 standard.

The elements themselves can also be defined. Two sample data element definitions (also in Lisp) are shown in FIG. 11 — element 93 is defined at 3100 and element 98 is defined at 3200. This definition allows the EDI system to verify whether the format of an input data element conforms to the data definition. The element definition includes the name of the element (e.g., 93, as shown at 3111), the data type (e.g., alpha-numeric, or AN, as shown at 3112), the minimum (3113) and maximum (3114) length, and the title (e.g., NAME, as shown in 3115). Next, the definition specifies the segments in which the element is used, as shown at 3120. Finally, the type and version of the EDI standard is specified in 3130.

As shown in the element definition for data element 98, a list of possible values, shown at 3220, can be included that is used to validate the data element values.

The e-language can be used to describe the document, segment, and element sets that make up EDI standards such as X12. New document, segment, or element types can be defined, or existing definitions can be modified. The e-language allows the user to express

the description in a way resembling the descriptions given in the EDI X12 and TDCC Standards books.

Each element definition includes a specified data type, such as the "AN" or alphanumeric data type specified for data element 93 at 3112 in FIG. 11. These data types are not hard coded in the e-language, but rather their format is defined by a regular expression, which is a notation well known in the art. Four sample data type definitions are shown in FIGS. 12A to 12C. The definition for a numeric data type with floating decimal is shown at 4100, the definition for a floating point decimal data type is shown at 4200, the definition for an alphanumeric data type is shown at 4300, and the definition for the different classes of data types found in a data element definition, and the verification generator associated with the class, is shown at 4400. For example, data types specified as "n2" will match the definition [nN] [0-9] at 4410, so that a pattern will be generated based on the X12-n-verification-generator. The verification generator accounts for the minimum and maximum lengths and scaling factors in developing the pattern. The generated pattern is later used to verify that data read from an EDI document matches the pattern.

These regular expressions are evaluated when the data element is read or assigned to a tree. This means that new data types defined by EDI standard-setting organizations can be incorporated into the system with little difficulty. Further, a sufficiently sophisticated user could introduce new data types.

An EDI tree can be constructed based on the document definition using one of two e-language primitives, READ EDI (or EDITOTREE) and MAKEDOC. Both primitives are based on the document definition as specified by its document identifier (such as 810), by the standard (such as X12), and by the version of the standard (such as version 2.2). The EDI system uses a hunt sequence associated with the version to resolve the appropriate segment and element definitions. This is a technique that reduces the amount of space required to store various versions of the standards. It is a defaulting scheme used so that standard specifications that remain unchanged from version to version are stored only once.

The READ EDI primitive reads a data stream according to a specified standard and document definition, constructs an EDI data tree structure, and populates the tree from the data stream so that it can be accessed using the EDI tree path notation. The MAKEDOC primitive prepares an empty EDI data tree structure based on a specified standard and document definition. As explained below, this empty tree can later be filled with assignment statements in a filemap structure. MAKEDOC creates the minimum bare tree corresponding to the specified document definition. However, additional nodes or intermediate branches may be needed to contain the data. When an assignment statement is executed, if the required intermediate branch or node does not yet exist in the tree, it is created by the assignment. Another primitive, WRITE EDI (or TREETOEDI) writes data that has been structured in an EDI data tree structure to an output stream in an EDI document format.

These primitives are implemented in the dynamic creation of data structures as specified by the standards. The document tree contains areas, which contain loop blocks or segment blocks, loop blocks in turn contain loop bodies or segment blocks, segment blocks contain segment bodies, and segment bodies contain data elements. To allow an NTE segment to float, any segment body can be given an NTE property. The user can refer to and manipulate the contents of the tree by using the flexible notation referred to above. A tree can have multiple occurrences of loops or segments. In traversing a tree, the specific occurrence required must be referred to. For example, in the 810 document definition shown in FIG. 9, there is a LOOP_N1 1110 defined in the HEADING area 1100 that can occur between 1 and 200 times. Inside the loop is an optional segment N2, shown at 1111, that can occur up to two times. To refer to the second occurrence of the N2 segment in the third occurrence of LOOP_N1, the tree path specification would be "tree\HEADING\LOOP_N1[3]\N2[2]."

These primitives can be used to take data from an input stream in either EDI or application format and put data into an output stream in either format. In the sample translation illustrated below, application input data is converted to EDI output data.

The translation of the sample invoice document shown in FIG. 7 to the EDI 810 format shown in FIG. 8 is implemented with a translation script. The EDI translation script used with these two documents is shown in FIGS. 13A to 13F.

This script illustrates the structure of an e-language program. A program written in the e-language consists of a series of program statements stored in a file with a structure similar to a Pascal program. The program can begin with array declarations, which can be one-dimensional or multi-dimensional. None are defined in this example. Next is the definitions section, in which functions, subroutines, records, and filemaps are defined. The function "the_date" is defined at 5110 in FIG. 13A, a filemap is defined at 5200 in FIGS. 13A to 13C, and several records are defined at 5300 in FIGS. 13C to 13F, with an individual record shown at 5310. The executable commands and statements follow, bounded by a BEGIN statement and an END statement, as shown at 5400 in FIG. 13F.

The FILEMAP statement, shown at 5200, maps the input data stream to a set of record assignments. These record assignments are listed in nested loops 5210 and 5211. Each record assignment assigns variables to tree path specifications. For example, in the "person" record assignment 5212, variable "p_name" is assigned to the tree path "tree\HEADING\LOOP_N1[1]\PER[1][2]."

Following the filemap definition is a set of record definitions, shown at 5220. These definitions pattern

match the data from the input file and assign them to the variables that have been assigned to tree paths in the record definition statements. In the command section 5400, a bare tree is created using the MAKEDOC function in line 5410, which specifies that the tree is to be an X12 810 invoice, using version 2.2 of X12. The bare tree is then populated from the standard input stream, STDIN, with the pattern-matched data in line 5420 using the MAPFILE function. Finally, in line 5430, data from the tree is output in EDI X12 format using the TREETOEDI function, using an "*" as an element separator, a "\n" as a segment separator, and an "@" as a subelement separator.

The operation of this script is illustrated with the sample input shown in FIG. 7 and the sample EDI output shown in FIG. 8. In record definition 5310, a search is conducted for the string "MAIL" by 5311 and 5312. This is found in the input document at 6110. Having found MAIL, the blanks following MAIL are skipped by line 5313. Then, the following fourteen characters of data, which in this case will include "Perry Lawrence," are read and assigned to the variable p_name in line 5314.

Line 5315 then searches for "GROUN." Then, the month ("NOV," read by line 5321), day ("27," read by line 5323), and year ("1989," read by line 5325) are read and assigned to the variables "mon," "day," and "year." These are concatenated by the function "the-date" defined at 5100 in FIG. 13A, which in turn was assigned to the variable "inv_date" in line 5313. Variable "inv_date" in turn is mapped to tree\HEADING\BIG[1][1] in line 5314 (i.e., the first element in the first occurrence of the BIG segment).

Although the ST (transaction set header) and SE (transaction set trailer) segments (shown at 1101 and 1321, respectively, in the document definition shown in FIG. 9) are mandatory because they envelope the document, they are not put into the EDI output document at this point in the process because they are not yet determined. Enveloping segments typically contain control information, which cannot be resolved until the translation step is completed. For example, the SE segment includes the number of segments in the document, which is not known until the document is finished.

As shown at 7111 in FIG. 8, the data is then output as "NOV271989" following the BIG segment identification. Similarly, line 5327 of FIG. 13D reads in the inventory number, 101903, assigning it to the variable "inv_num," which in turn is mapped by line 5315 to tree\HEADING\BIG[1][2], from which it is output at 7112. This process is followed to read the remainder of the required information from the input document and map it to the tree and thence to the output document.

The inverse of the above operation can also be performed. An EDI document can be converted into data records in a format readable by an application. In such a translation, a tree corresponding to the EDI invoice is created and populated using the READ EDI primitive. A series of assignment statements are used to assign data from the tree to fields in data records. When a data record is completed, it is output to the application by the PUTREC command. A sample of a script for translating an EDI document into an application document is shown in FIGS. 14A to 14F.

In the commands in FIGS. 14A to 14D, such as "record header" at 13110, the fields for each record of data required by the application are defined with picture statements. At 13200 in FIG. 14D, the READ EDI primitive is used to read the document "podoc" (purchase order document) and create a tree structure populated with the data from the EDI document in the input data stream. The script next includes a series of assignment statements, as at 13310, in which data from nodes of the tree are assigned to the fields. When the fields in a record are filled, the record is output to the output stream with the PUTREC command, as at 13400. This process continues for each of the records.

A translation from one EDI format to another EDI format can be performed as follows. First, READ EDI is used to create and populate a first tree corresponding to the input document. MAKEDOC is used to create a bare second tree corresponding to the desired output document. A series of assignment statements are then used to populate the bare second tree with data from the first tree. WRITE EDI then creates an EDI document from the populated second tree.

Similarly, the system can translate a document from a first application format to a second application format. An intermediate tree structure may not be used in this case. The document is first pattern recognized. The pattern recognized data is then assigned to the appropriate records for the second document format, and the records output to the second application using PUTREC. A tree may also be used to represent application transactions.

Enveloping Work Center

Outbound EDI documents are created by the translator process. These documents are usually created as a result of processing an inbound application transaction. Although the outbound document contains a complete address, the facility used for communicating the document may not be ready. There is therefore a need to hold these documents until the communication facility is ready. The process of grouping EDI documents into a "package" that is to be sent through an EDI communication facility is termed "enveloping."

As described above, the EDI enveloping scheme employs the concept of an interchange, which is a unit of communication from a given sender to a specific receiver comprising a package of mail containing multiple functional groups. A functional group is a grouping of EDI documents that correspond to a theoretical organizational function, such as the purchasing department. Each document is a single business document expressed in EDI.

The system uses a hierarchical file structure to store the EDI documents that are ready to be communicated. The structure closely follows the EDI enveloping scheme. The hierarchy runs from communication facility as the highest level, to network interface, receiver, sender, functional group, and document. This hierarchical structure is also used to store application transactions. Outbound application transactions are stored in a similar hierarchy, with communication facility at the top, followed by application interface and application transaction.

Thus, a file produced by the translator and its accompanying routing form is moved into a depository where the enveloper can find the file when required. The translator uses the node, file type (EDI, non-EDI, or text) network or application interface, and other information on the routing form to determine where a file should be placed. It also consults a table that indicates what sort of interchange is used for a given trading partner, functional group, and node. This file structure, corresponding to the hierarchy described above, is shown graphically in FIG. 6. TP denotes a trading partner, INT interchange method, and FG functional group. In the illustrated embodiment, the file is physically located on the fixed disk of the microcomputer on which the system is operated.

When the communication facility is ready, it makes a request to the enveloper specifying the communication facility and either the application interface or the network interface. A network interface is an interface to an EDI transfer or user agent. This means that the packages communicated through this interface must follow the EDI packaging scheme described above. An application interface is an application transfer or user agent. This means that the packages communicated through this interface pertains to a more simplified packaging. The procedure is to concatenate all transactions and present them for transmission. Optionally, the transactions may also be surrounded by job control language to create a job stream to be transmitted to a host.

When it receives a request from a facility client, the enveloper takes files from the directories into which they were placed by the depositor. The enveloper locates the appropriate directory based on the identity of the node and facility being served and the type of file content. Files within a given functional group directory are wrapped in a functional group envelope, which is in turn wrapped in an interchange envelope. Values for data elements in the functional group and interchange envelopes are obtained or generated. When the interchange is complete, it is sent to the facility client in the communications interface.

For application data files output by the translation work center, the enveloper first batches together the documents into a single set for use by the pertinent application. In the illustrated embodiment, the enveloper packages the data for the host (attaching job control language, for example) and transmits the packaged data through the communications interface to the job entry system of the host. In an alternate embodiment where the application and the EDI translation system operate on the same machine, the documents are grouped into a single file and placed in a file location agreed to by the application and EDI system, and the application is then triggered to execute the file.

Text files are sent directly to the facility client without processing. Sender and receiver data are embedded in application files. This makes the information available to the receiver and to the user's enveloping script.

Claims

1. A method for translating electronic data in a computer system (10) from a first format to a second format, comprising the steps of:

   (a) determining (1) which one of a plurality of communication protocols to utilize to receive data as a function of a communication process used to transmit the data to the computer system (10);
   (b) receiving (1) input data in the first format, the data comprising a plurality of data components;

   characterized by

   (c) assigning (2) a script name (8110, 8120, 8210, 8310, 8410, 9120, 9110, 9210) to the received data to identify a de-enveloping procedure that will be used to separate the plurality of data components of the received data into individual data components, the de-enveloping procedure identified being dependent on the communication process used to transmit the input data to the computer system (10);
   (d) dividing (2) the received data into individual data components by executing the identified de-enveloping procedure;
   (e) translating (3) the individual data components from the first format into the second format which is chosen to be compatible with a desired destination for the data; and
   (f) arranging (4) the individual data components into a package so that the package is available for transmission at any time by the computer system (10) to the desired destination.

2. The method of claim 1, wherein said unit of work is divided into pages, said data components comprise application data components, and said dividing step comprises the steps of:

   a. marking a beginning of a data component;
   b. parsing through said unit of work;

c. pattern matching data in said pages and extracting an identifier from each of said pages;
d. marking an end of a data component when said identifier changes; and
e. dividing out as a data component the data between said marked beginning and end.

3. The method of claim 1, wherein said script is defined in a programming language, said programming language incorporating a group of translating instructions corresponding to several translating steps in the flow of translating, including a pattern matching procedure for extracting data from a data stream, a data tree structure creation procedure, and a mapping procedure for mapping extracted data to a tree structure and mapping a data tree structure to an output stream.

4. The method of claim 3, wherein said programming language uses a relational model to represent data to facilitate processing by structured query language commands.

5. The method of claim 3, wherein said programming language is executed through an interpreter.

6. The method of claim 1, wherein said translating step comprises the steps of:

a. executing a first script to identify, classify, organize, route, and determine the translation requirements of said plurality of data components in said first format; and
b. executing a second script to transform said plurality of data components in said first format to said second format.

7. The method of claim 6, wherein said scripts are executed by an interpreter.

8. The method of claim 6, wherein said scripts are executed by a compiler.

9. The method of claim 6, wherein said transforming step comprises the steps of:

a. constructing a data tree structure having a plurality of branches and a plurality of nodes to represent a document or transaction;
b. mapping one of said plurality of data components in said first format onto said data tree structure;
c. mapping data in said data tree structure into said plurality of data components in said second format.

10. The method of claim 1, wherein said individual data components in said second format are maintained in a depository until they are transmitted to said destination.

11. The method of claim 9, wherein said data tree structure is defined by specifications of an electronic data interchange document standard.

12. The method of claim 11, wherein said specification comprise segment definitions and element definitions and wherein said step of constructing a data tree structure comprises the steps of:

a. identifying a specification;
b. defining a plurality of nodes, said nodes corresponding to said segment and element definitions; and
c. developing a path specification for accessing said nodes.

13. The method of claim 12, wherein said step of identifying a specification comprises the steps of:

a. identifying a specification identifier number;
b. identifying an EDI standard;
c. identifying an EDI standard version number; and
d. using a hunt sequence associated with said version to resolve said segment and element definitions.

14. The method of claim 12, wherein said element definitions have associated data types, said data types being defined by a regular expression, said regular expression being computed when said element definition is read.

15. The method of claim 9, wherein said step of mapping to said data tree comprises the steps of:

a. defining a plurality of data records, each of said records having one or more fields;
b. parsing said one of said plurality of data components;
c. matching data in said one of said plurality of data components to said fields in said record definitions; and
d. binding said fields to said matching data.

16. The method of claim 9, wherein said step of mapping from said tree structure comprises the steps of:

a. determining the identity of delimiters to be used to delimit said data;
b. traversing said data tree in a depth first order; and
c. formatting each of said nodes and placing said delimiters.

17. A system for translating electronic data from a first format to a second format, comprising:

(a) a network; and
(b) a computer coupled to the network for receiving data transmitted from the network, said computer including:

(b1) a communication interface (1) including a driver to accept data in the first format, the data comprising a plurality of data components, and a script (8110, 8120, 8210, 8310, 8410) which determines a communication protocol to utilize to accept the data as a function of a communication process used to transmit the data to the communication interface,
(b2) a de-enveloper (2) coupled to the communication interface (1), said de-enveloper (2) including

- means for receiving the data from the communication interface (1),
- means for assigning a script name (8110, 8120, 8210, 8310, 8410, 9120, 9110, 9210) to the received data to identify a de-enveloping procedure that will be used to separate the received data into individual data components, the identified de-enveloping procedure being dependent on the communication process used to transmit the data to the communication interface (1), and
- means for dividing the received data into individual data components by executing the identified de-enveloping procedure,

(b3) a translator (3) coupled to the de-enveloper (2), said translator (3) manipulating the individual data components from the first format into the second format which is chosen to be compatible with a desired destination for the data, and
(b4) an enveloper (4) coupled to the translator (3) and the communication interface (1), said enveloper (4) grouping the translated individual data components into a package so that the package is available for transmission at any time by the communication interface (1) to the desired destination.

18. The system according to claim 17, wherein said means for assigning a script name further includes:

means for dividing the received data into individual data components;
means for identifying sender, receiver and transaction type; and
means for determining the type of transformation needed and destination of the transformed individual data components.

Patentansprüche

1. Verfahren zum Übersetzen elektronischer Daten in einem Computersystem (10) von einem ersten Format in ein zweites Format, die folgenden Schritte aufweisend:

(a) Entscheiden (1), welches von einer Anzahl von Kommunikationsprotokollen zu verwenden ist, um Daten als eine Funktion eines Kommunikationsprozesses zu empfangen, der verwendet wird, um die Daten an das Computersystem (10) zu übertragen;
(b) Empfangen (1) von Eingangsdaten in dem ersten Format, wobei die Daten eine Anzahl von Datenkomponenten aufweisen;

gekennzeichnet durch

(c) Zuweisen (2) eines Skriptnamens (8110, 8120, 8210, 8310, 8410, 9120, 9110, 9210) zu den empfangenen Daten, um eine Ent-Umhüllungsprozedur zu identifizieren, die verwendet wird, um die Anzahl von Datenkomponenten der empfangenen Daten in individuelle Datenkomponenten zu separieren, wobei die identifizierte Ent-Umhüllungsprozedur von dem Kommunikationsprozeß abhängt, der verwendet wird, um die Eingangsdaten an das Computersystem (10) zu übertragen;
(d) Teilen (2) der empfangenen Daten in individuelle Datenkomponenten durch Ausführen der identifizierten Ent-Umhüllungsprozedur;
(e) Übersetzen (3) der individuellen Datenkomponenten von einem ersten Format in das zweite Format, welches gewählt ist, um mit einer gewünschten Bestimmung für die Daten kompatibel zu sein; und
(t) Anordnen (4) der individuellen Datenkomponenten in ein Paket, derart, daß das Paket zur Übertragung zu jeder beliebigen Zeit durch das Computersystem (10) an die gewünschte Bestimmung verfügbar ist.

2. Verfahren gemäß Anspruch 1, bei welchem die Arbeitseinheit in Seiten aufgeteilt wird, wobei die Datenkomponenten Anwendungsdatenkomponenten aufweisen und wobei der Aufteilungsschritt die folgenden Schritte aufweist:

a. Markieren eines Beginns einer Datenkomponente;
b. Untergliedern der Arbeitseinheit;
c. Muster-Gleichheitsüberprüfung von Daten auf den Seiten und Extrahieren eines Kennzeichners von jeder der Seiten;
d. Markieren eines Endes einer Datenkomponente, wenn der Kennzeichner wechselt; und
e. Herausteilen als eine Datenkomponente der Daten zwischen dem markierten Beginn und Ende.

3. Verfahren gemäß Anspruch 1, bei welchem das Skript in einer Programmiersprache definiert ist, wobei die Programmiersprache eine Gruppe von Übersetzungsbefehlen beinhaltet, die mehreren Übersetzungsschritten in dem Fluß des Übersetzens entsprechen, inklusive einer Muster-Gleichheitsüberprüfungsprozedur zum Extrahieren von Daten aus einem Datenstrom, einer Datenbaumstruktur-Erzeugungsprozedur und einer Abbildungsprozedur zum Abbilden extrahierter Daten in eine Baumstruktur und zum Abbilden einer Datenbaumstruktur in einen Ausgangsstrom.

4. Verfahren gemäß Anspruch 3, bei welchem die Programmiersprache ein relationales Modell verwendet, um Daten zu repräsentieren, um Verarbeitung durch strukturierte Abfragesprache-Befehle zu ermöglichen.

5. Verfahren gemäß Anspruch 3, bei welchem die Programmiersprache durch einen Interpreter ausgeführt wird.

6. Verfahren gemäß Anspruch 1, bei welchem der Übertragungsschritt die folgenden Schritte aufweist:

a. Ausführen eines ersten Skripts, um die Übertragungserfordernisse der Anzahl von Datenkomponenten in dem ersten Format zu identifizieren, klassifizieren, organisieren, routen und bestimmen; und
b. Ausführen eines zweiten Skripts, um die Anzahl von Datenkomponenten in dem ersten Format in das zweite Format zu transformieren.

7. Verfahren gemäß Anspruch 6, bei welchem die Skripte durch einen Interpreter ausgeführt werden.

8. Verfahren gemäß Anspruch 6, bei welchem die Skripte durch einen Compiler ausgeführt werden.

9. Verfahren gemäß Anspruch 6, bei welchem der Transformationsschritt die folgenden Schritte aufweist:

a. Konstruieren einer Datenbaumstruktur mit einer Anzahl von Zweigen und einer Anzahl von Knoten, um ein Dokument oder eine Transaktion darzustellen;
b. Abbilden einer von der Anzahl von Datenkomponenten in dem ersten Format auf die Datenbaumstruktur;
c. Abbilden von Daten in der Datenbaumstruktur in die Anzahl von Datenkomponenten in dem zweiten Format.

10. Verfahren gemäß Anspruch 1, bei welchem die individuellen Datenkomponenten in dem zweiten Format in einer Ablage gehalten werden, bis sie an das Ziel übertragen werden.

11. Verfahren gemäß Anspruch 9, bei welchem die Datenbaumstruktur durch Spezifikationen eines elektronischen Datenaustausch-Dokumentenstandards definiert ist.

12. Verfahren gemäß Anspruch 11, bei welchem die Spezifikation Segment-Definitionen und Element-Definitionen aufweist und bei welchem der Schritt des Konstruierens einer Datenbaumstruktur die folgenden Schritte aufweist:

a. Identifizieren einer Spezifikation;
b. Definieren einer Anzahl von Knoten, wobei die Knoten den Segment- und den Element-Spezifikationen entsprechen; und
c. Entwickeln einer Pfad-Spezifikation zum Zugreifen auf die Knoten.

13. Verfahren gemäß Anspruch 12, bei welchem der Schritt des Identifizierens einer Spezifikation die folgenden Schritte aufweist:

a. Identifizieren einer Spezifikations-Kennzeichnerzahl;
b. Identifizieren eines EDI-Standards;
c. Identifizieren einer EDI-Standard-Versionszahl; und
d. Verwenden einer Suchsequenz, die der Version zugeordnet ist, um die Segment- und Element-Definitionen aufzulösen.

14. Verfahren gemäß Anspruch 12, bei welchem die Element-Definitionen zugeordnete Datentypen aufweisen, wobei die Datentypen durch einen Regel-Ausdruck definiert sind, wobei der Regel-Ausdruck berechnet wird, wenn die Element-Definition gelesen wird.

15. Verfahren gemäß Anspruch 9, bei welchem der Schritt des Abbildens auf den Datenbaum die folgenden Schritte aufweist:

a. Definieren einer Anzahl von Datensätzen, wobei jeder der Sätze ein oder mehrere Felder aufweist;
b. Untergliedern der einen der Anzahl von Datenkomponenten;
c. Gleichheitsüberprüfung von Daten in der einen der Anzahl von Datenkomponenten mit den Feldern in den genannten Satz-Definitionen; und
c. Binden der Felder an die Gleichheitsüberprüfungs-Daten.

16. Verfahren gemäß Anspruch 9, bei welchem der Schritt des Abbildens von der Baumstruktur die folgenden Schritte aufweist:

a. Bestimmen der Identität von Abgrenzern, die verwendet werden, um die Daten abzugrenzen;
b. Durchschreiten des Datenbaumes in einer In-Die-Tiefe-Zuerst-Reihenfolge; und
c. Formatieren jedes der Knoten und Plazieren der Abgrenzer.

17. System zum Übersetzen elektronischer Daten von einem ersten Format in ein zweites Format, aufweisend:

(a) ein Netzwerk; und
(b) einen Computer, der mit dem Netzwerk gekoppelt ist, zum Empfangen von von dem Netzwerk übertragenen Daten, wobei der Computer aufweist:

(b1) eine Kommunikations-Schnittstelle (1) mit einem Treiber, um Daten in dem ersten Format zu akzeptieren, wobei die Daten eine Anzahl von Datenkomponenten und ein Skript (8110, 8120, 8210, 8310, 8410) aufweisen, welches das Kommunikationsprotokoll bestimmt, welches zu verwenden ist, um die Daten zu akzeptieren, als eine Funktion eines Kommunikationsprozesses, der verwendet wird, um die Daten an die Kommunikations-Schnittstelle zu übertragen,
(b2) einen Ent-Umhüller (2), gekoppelt mit der Kommunikations-Schnittstelle (1), wobei der Ent-Umhüller (2) aufweist:

- Mittel zum Empfangen der Daten von der Kommunikations-Schnittstelle (1),
- Mittel zum Zuweisen eines Skriptnamens (8110, 8120, 8210, 8310, 8410, 9120, 9110, 9210) an die empfangenen Daten, um die Ent-Umhüllungsprozedur zu identifizieren, die verwendet wird, um die empfangenen Daten in individuelle Datenkomponenten zu separieren, wobei die identifizierte Ent-Umhüllungsprozedur von dem Kommunikationsprozeß abhängt, der verwendet wird, um die Daten an die Kommunikations-Schnittstelle (1) zu übertragen, und
- Mittel zum Aufteilen der empfangenen Daten in individuelle Datenkomponenten durch Ausführen der identifizierten Ent-Umhüllungsprozedur,

(b3) einen Übersetzer (3) gekoppelt mit dem Ent-Umhüller (2), wobei der Übersetzer (3) die individuellen Datenkomponenten von dem ersten Format in das zweite Format manipuliert, welches gewählt wird, um mit einer gewünschten Bestimmung für die Daten kompatibel zu sein, und
(b4) einen Ent-Umhüller (4), gekoppelt mit dem Übersetzer (3) und der Kommunikations-Schnittstelle (1), wobei der Ent-Umhüller (4) die übertragenen individuellen Datenkomponenten in ein Paket gruppiert, derart, daß das Paket zur Übertragung zu jeder beliebigen Zeit durch die Kommunikations-Schnittstelle (1) an die gewünschte Bestimmung verfügbar ist.

18. System gemäß Anspruch 17, bei welchem die Mittel zum Zuweisen eines Skriptnamens weiter aufweisen:

Mittel zum Aufteilen der empfangenen Daten in individuelle Datenkomponenten;
Mittel zum Identifizieren von Sender, Empfänger und Transaktionstyp; und
Mittel zum Bestimmen des Typs der benötigten Transformation und der Bestimmung der transformierten individuellen Datenkomponenten.

Revendications

1. Procédé pour traduire des données électroniques dans un système d'ordinateur (10) à partir d'un premier format en un second format, comprenant des étapes suivantes:

(a) déterminer (1) lequel d'une pluralité des protocoles de communication est à utiliser pour recevoir des données en fonction d'un processus de communication utilisé pour transmettre les données au système d'ordinateur (10);
(b) recevoir (1) des données d'entrée d'un premier format, les données comprenant une pluralité de composantes de données;

charactérisé par les étapes suivantes:

(c) attribuer (2) un nom de scripte (8110, 8120, 8210, 8310, 8410, 9120, 9110, 9210) aux données reçues pour identifier une procédure de développement qui sera utilisée pour séparer la pluralité des composantes de données des donnés reçues en des composantes de données individuelles, la procédure de développement identifiée étant dépendante du processus de communication utilisé pour transmettre les données d'entrée au système d'ordinateur (10);
(d) diviser (2) les données reçues en composantes de données individuelles en exécutant la procédure de développement identifiée;
(e) traduire (3) les composantes de données individuelles à partir du premier format en second format qui est choisi pour être compatible avec la destination désirée pour les données; et
(f) arranger (4) les composantes de données individuelles dans un paquet afin que le paquet soit disponible pour la transmission à chaque instant par le système d'ordinateur (10) à la destination désirée.

2. Le procédé de la revendication 1, dans lequel ladite unité de travail est divisée en pages, lesdites composantes de données comprenant des composantes de données d'application, ladite étape de division comprenant les étapes suivantes:

a. marquer un début d'une composante de données;
b. analyser ladite unité de travail;
c. mettre en correspondance des données sur lesdites pages et extraire un identificateur de chacune desdites pages;
d. marquer une fin d'une composante de données quand ledit identificateur change; et
e. sortir comme une composante de données les données entre ledit début et ladite fin marqués.

3. Le procédé de la revendication 1, dans lequel ledit scripte est défini dans un langage de programmation, ledit langage de programmation incorporant un groupe d'instruction de traduction correspondant à plusieurs étapes de traduction dans le flux de traduction, incluant une procédure de mise en correspondance pour extraire des donnés d'un flux de données, une procédure de création d'une arborescence de données, et d'une procédure de représentation pour représenter des données extraites en une arborescence et pour représenter une arborescence de données en un flux de sorti.

4. Le procédé de la revendication 3, dans lequel ledit langage de programmation utilise un modèle relationnel pour représenter des données afin de faciliter le traitement par des commandes de langage d'interrogation structurées.

5. Le procédé de la revendication 3, dans lequel ledit langage de programmation est exécuté par un interprétateur.

6. Le procédé de la revendication 1, dans lequel ladite étape de traduction comprend les étapes suivantes:

a. exécuter un premier scripte afin d'identifier, classifier, organiser, router et déterminer les besoins de traduction de ladite pluralité des composantes de données dans le premier format: et
b. exécuter un second scripte afin de transformer ladite pluralité des composantes de données dans ledit premier format en ledit second format.

7. Le procédé de la revendication 6, dans lequel lesdits scriptes sont exécutés par un interprétateur.

8. Le procédé de la revendication 6, dans lequel lesdits scriptes sont exécutés par un compileur.

9. Le procédé de la revendication 6, dans lequel ladite étape de transformation comprends les étapes suivantes:

a. construire une arborescence de données ayant une pluralité de branches et une pluralité de noeuds pour représenter un document ou une transaction;
b. représenter une de ladite pluralité de composantes de données en ledit premier format en ladite arborescence de données;
c. représenter des données dans ladite arborescence de données en ladite pluralité de composantes de données dans ledit second format.

10. Le procédé de la revendication 1, dans lequel lesdites composantes de données individuelles dans ledit second format sont maintenues dans un dépôt jusqu'à ce qu'elles soient transmises à ladite destination.

11. Le procédé de la revendication 9, dans lequel ladite arborescence de données est définie par des spécifications d'un standard de documents d'échange de données électroniques.

12. Le procédé de la revendication 11, dans lequel ladite spécification comprend des définitions de segments et des définitions d'éléments et dans lequel ladite étape de construction d'une arborescence de

données comprend les étapes suivantes:

a. identifier une spécification;
b. définir une pluralité de noeuds, lesdits noeuds correspondant auxdites définitions de segments et d'éléments; et
c. développer une spécification de chemin pour accéder auxdits noeuds.

13. Le procédé de la revendication 12, dans lequel ladite étape d'identification d'une spécification comprend les étapes suivantes:

a. identifier un numéro d'identificateur des spécification;
b. identifier un standard EDI;
c. identifier un numéro de version d'un standard EDI; et
d. utiliser une séquence de recherche associée à ladite version pour dissoudre lesdites définitions de segments et d'éléments.

14. Le procédé de la revendication 12, dans lequel lesdites définitions d'éléments ont des types de données associées, lesdits types de données étant définis par une expression de règle, ladite expression de règle étant calculée quand ladite définition d'éléments est lue.

15. Le procédé de la revendication 9, dans lequel ladite étape de représentation à ladite arborescence de données comprenant les étapes suivantes:

a. définir une pluralité d'articles de données, chacun desdits articles ayant un ou plusieurs champs;
b. analyser ladite une de ladite pluralité de composantes de données;
c. mettre en correspondance des données dans ladite une de ladite pluralité de composantes de données avec lesdits champs dans lesdites définitions d'articles; et
d. lier lesdits champs auxdites données comparées.

16. Le procédé de la revendication 9, dans lequel ladite étape de représenter de ladite arborescence comprend les étapes suivantes:

a. déterminer l'identité de délimiteurs à être utilisée pour délimiter lesdites données;
b. traverser ledit arbre de données dans l'ordre "profondeur d'abord"; et
c. formater chacun desdits noeuds et placer lesdits délimiteurs.

17. Système pour traduire des données électroniques d'un premier format en un second format, comprenant:

(a) un réseau; et
(b) un ordinateur couplé aux réseaux pour recevoir des données transmises du réseau, ledit ordinateur comprenant:

(b1) une interface (1) de communication ayant un gestionnaire afin d'accepter des données dans le premier format, les données ayant une pluralité de composantes de données, et un scripte (8110, 8120, 8210, 8310, 8410) qui déterminent le protocole de communication à utiliser pour accepter les données en fonction d'un processus de communication utilisé pour transmettre les données à l'interface de communication,
(b2) un développeur (2) couplé à l'interface de communication (1), le développeur (2) ayant

- moyens pour recevoir les données de l'interface de communication (1),
- moyens pour assigner un nom de scripte (8110, 8120, 8210, 8310, 8410, 9120, 9110, 9210) aux données reçues afin d'identifier une procédure de développement qui sera utilisée pour séparer les données reçues en composantes de données individuelles, la procédure de développement identifiée étant dépendant du processus de communication utilisé pour transmettre les données à l'interface de communication (1), et
- moyens pour diviser les données reçues en composantes de données individuelles pour exécuter la procédure de développement identifiée,

(b3) un traducteur (3) couplé au développeur (2), ledit traducteur (3) manipulant les composantes de données individuelles dudit premier format en le second format qui est choisi à être compatible avec la destination désirée pour les données, et
(b4) un envelopppeur (4) couplé au traducteur (3) et à l'interface de communication (1), ledit enveloppeur (4) groupant les composantes de données individuelles traduites dans un paquet afin que le paquet soit disponible pour transmission à chaque instant par l'interface de communication (1) à la destination désirée.

18. Le système de la revendication 17, dans lequel lesdits moyens pour assigner un nom de scriptes com-

prend en plus:

des moyens pour diviser les données reçues en composantes de données individuelles;
des moyens pour identifier l'émetteur, le récepteur et le type de transaction; et
des moyens pour déterminer le type de transformation nécessaire et la destination des composantes de données individuelles transformées.

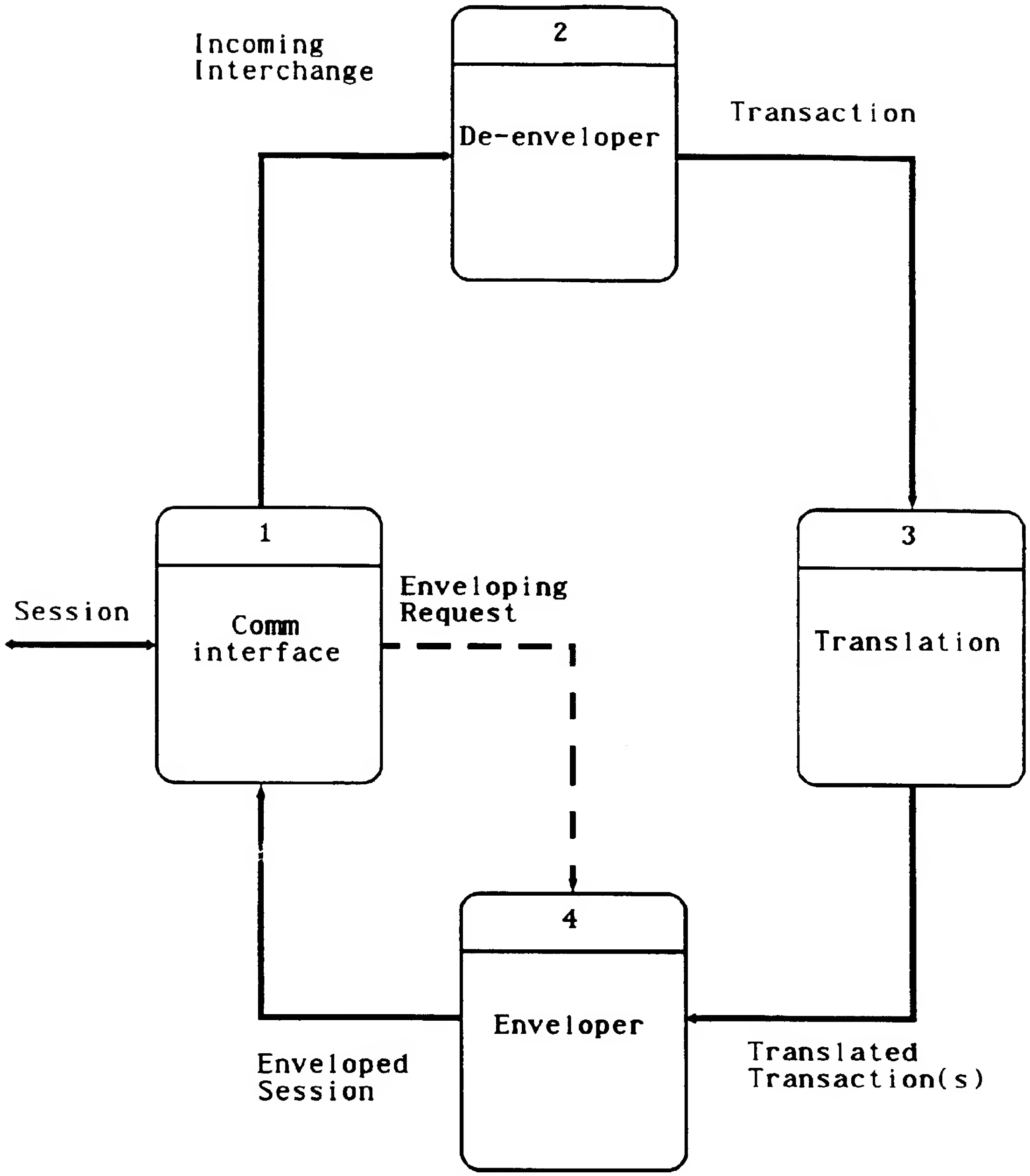
Incoming
Interchange
2
De-enveloper
Transaction
1
Session
Comm
interface
Enveloping
Request
3
Translation
4
Enveloper
Enveloped
Session
Translated
Transaction(s)

FIG. 1

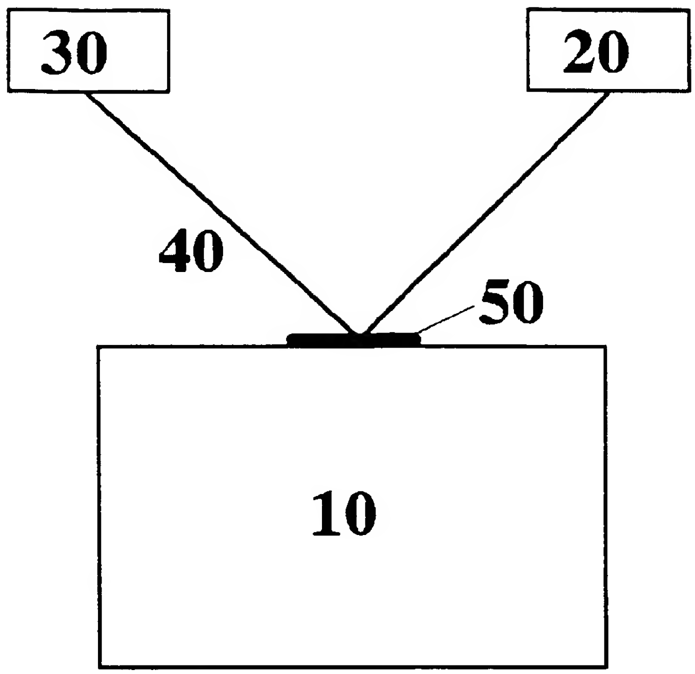

FIG. 2

```
                 8110          8120
     stream = Envelope   ("ATTMAIL")
8210 SendUnixMail (stream)
     vfList = ReadUnixMail ("ATTMAIL");
                 8510

     for each in vfList
       begin
8410   MakeMailInterchange
       end
```

FIG. 3

```
     {
      ' read and discard text up to first occurrence of BS or ISA
9110  cursor = OpenInterchange (stdin,             "ISA")
9210  xcan = Deenvelope (X12,cursor)        ' extract virtual file

      while not null (xacn) begin
        route (xacn)                  ' send virtual file to servicer
        xacn = Deenvelope(X12,cursor);    ' and repeat
        }
     }
```

FIG. 4

```
(defun convmonth (m d y)
  (catch '$retval_convmonth
    (setq mdy (concat m d))
    (setq mdy (concat md y))
    (throw '$retval_convmonth mdy)))

10110 (setq invoice '(
                 (loop
                   ((record header)
                     (progn
                       (printf "%s" "remit to: ")
                       (printf "%x" contact)
                       (printf "\n")))
                   ((record unused))
                     (setq invdate (convmonth month day year))
                     (progn
                       (printf "%s" "date: ")
                       (printf "%s" invdate)
                       (printf "\n")))
                   ((record unused))
           10120 ((record terms)
                     (progn
                       (printf "%s" "cust. number:")
           10121       (printf "[%s]" custnum)
                       (printf "\n")
                       (printf "%s" "terms;")
                       (printf "%s" termper)
                       (printf "%s" "%")
                       (printf "%s" termdays)
                       (printf "\n")))
                   ((record unused))
                   ((record names)
                     (progn
                       (printf "%s" "bill to:")
                       (printf "%s" billname)
                       (printf "\n")))
```

FIG. 5A

```
((record streets)
(progn
     (printf "%s" "street:")
     (printf "%" billstrt)
     (printf "\n")))
((record csz)
  (progn
     (printf "%s" "city,state,zip:")
     (printf "%s" billcsz)
     (printf "\n")))
((record unused)
  (progn
     (printf "%s" "unsused")
     (printf "\n")))
((record unused)
  (setq n 1))
(loop
((record detail)
  (if (> dozens 0)
     (setq itemun (* dozens units))
     (setq itemun units))
(progn
  (printf "%s" item)
  (printf "%s" dettot)
  (printf "%s" price)
  (printf "%s" units)
  (printf "%s" dozens)
  (printf "\n"))))
((record unused)
  (progn
  (printf "%s" "unsused")
  (printf "\n")))
((record unused)
  (progn
     (printf "%s" "unsused")
     (printf "\n")))
((record unused)
  (progn
     (printf "%s" "unsused")
     (printf "\n")))
```

FIG. 5B

```
                  ((record total1)
                     (progn
                        (printf "%s" merchtot)
                        (printf "\n")))
                  ((record total2)
                     (progn
                        (printf "%s" nontot)
                        (printf "\n")))
    10300   10310  ((record total3)
                     (progn
                10310  (mark_end)
    10400   10410      (setq vf (do_split))
    ;; every time a record of the format total13 is found set the
    ;; ending mark, and split.
                10420  (set_app_normal_receiver vf custnum)
    ;; determine the receiver using the data element
          custnum found
    ;; in the terms record earlier in the scanning.
                10430 (mark_start)
    ;; reset the start marker
                        (printf "%s" (-tottot nontot))
                        (printf "\n")) ))))
10440 (setq header '(
                           (filler (x 0) nil)
                           (filler (x 4) "MAIL")
                           (filler (b 5) nil)
                           (contact (x 15) nil)
                           (filler (x 0 nil)
                           (filler (x 9) "INVOICE #")
                           (filler (x 1) "\n")))

      (setq info '(
                           (ponum (x 6) nil)
                           (filler (x 33) nil)
                           (showeth (x 20) nil)
                           (month (x 3) nil)
                           (filler (b 1) nil)
                           (day (x 2) nil)
```

FIG. 5C

```
                (filler (b 1) nil)
                (year (x 4) nil)
                (filler (x 3) nil)
                (invnum (x 0) nil)
                (filler (x 1) "\n")))

(setq names '(
                (filler (x 0) nil)
                (filler (x 5) "BILL")
                (billname (x 25) nil)
                (filler (x 0) nil)
                (filler (x 5) "SHIP")
                (shipname (x 9) nil)
                (filler (b 0) nil)
                (filler (x 1) "\n")))

(setq detail '(
                (item (x 8) nil)
                (filler (x 14) nil)
                (filler (x 1) " ")
                (filler (x 30) nil)
                (dozens (n "9") nil)
                (filler (b 1) nil)
                (units (n "z9") nil)
                (filler (b 7) nil)
                (price (n "z") nil)
                (filler (b 7) nil)
                (detto (n "zz9") nil)
                (filler (x 0) nil)
                (filler (x 1) "\n")))

(setq total1 '(
                (filler (x 0) nil)
                (filler (x 18) "MERCHANDISE TOTAL-")
                (filler (x 7) nil)
                (merchtot (n "zz9") nil)
                (filler (x 1) "\n")))
```

FIG. 5D

```
(setq total2 '(
                (filler (x 0) nil)
                (filler (x 16) "NON-MERCH TOTAL-")
                (filler (x 7) nil)
                (nontot (n "zzz9") nil)
                (filler (x 1) "\n")))

(setq total3 '(
                (filler (x 16) nil)
                (duedte (x 11) nil)
                (filler (x 0) nil)
                (filler (x 14) "INVOICE TOTAL-")
                (filler (x 7) nil)
                (tottot (n "zz9") nil)
                (filler (x 1) "\n")))

(setq streets '(
                (filler (b 0) nil)
                (filler (x 4) "TO  ")
                (billstrt (x 17) nil)
                (filler (x 0) nil)
                (filler (x 4) "TO  ")
                (shipstrt (x 19) nil)
                (filler (x 0) nil)
                (filler (x 1) "\n")))

(setq csz '(
                (filler (b 0) nil)
                (billcsz (x 29) nil)
                (filler (b 0) nil)
                (shipcsz (x 10) nil)
                (filler (x 0) nil)
                (filler (x 1) "\n")))

(setq terms '(
                (custnum (x 6) nil)
                (filler (x 55) nil)
                (termper (n "9") nil)
                (filler (x 2) nil)
```

FIG. 5E

```
                    (termdays (x 2) nil)
                    (filler (x 0) nil)
                    (filler (x 1) "\n")))

    (setq unused '(
                    (filler (x 0) nil)
                    (filler (x 1) "\n")))

  ;; Sets the start marker at the begining of the data stream

10520 (setq vfl (split stdin invoice))

  ;; Asks that each invoice to split and keep a list in the
  ;; variable vfl - the flemap to be used is "invoice" and the
  ;; data to be matched is in "stdin"

10600  10610  (while vfl
   10620 (setq vf (car vfl))
   10630 (printf "deenv: routing to translator\n")

   10640  (route_app_to_translator vf)
     ;; each invoice document is routed to the translator
        work center

     (setq vfl (cdr vfl)))
```

FIG. 5F

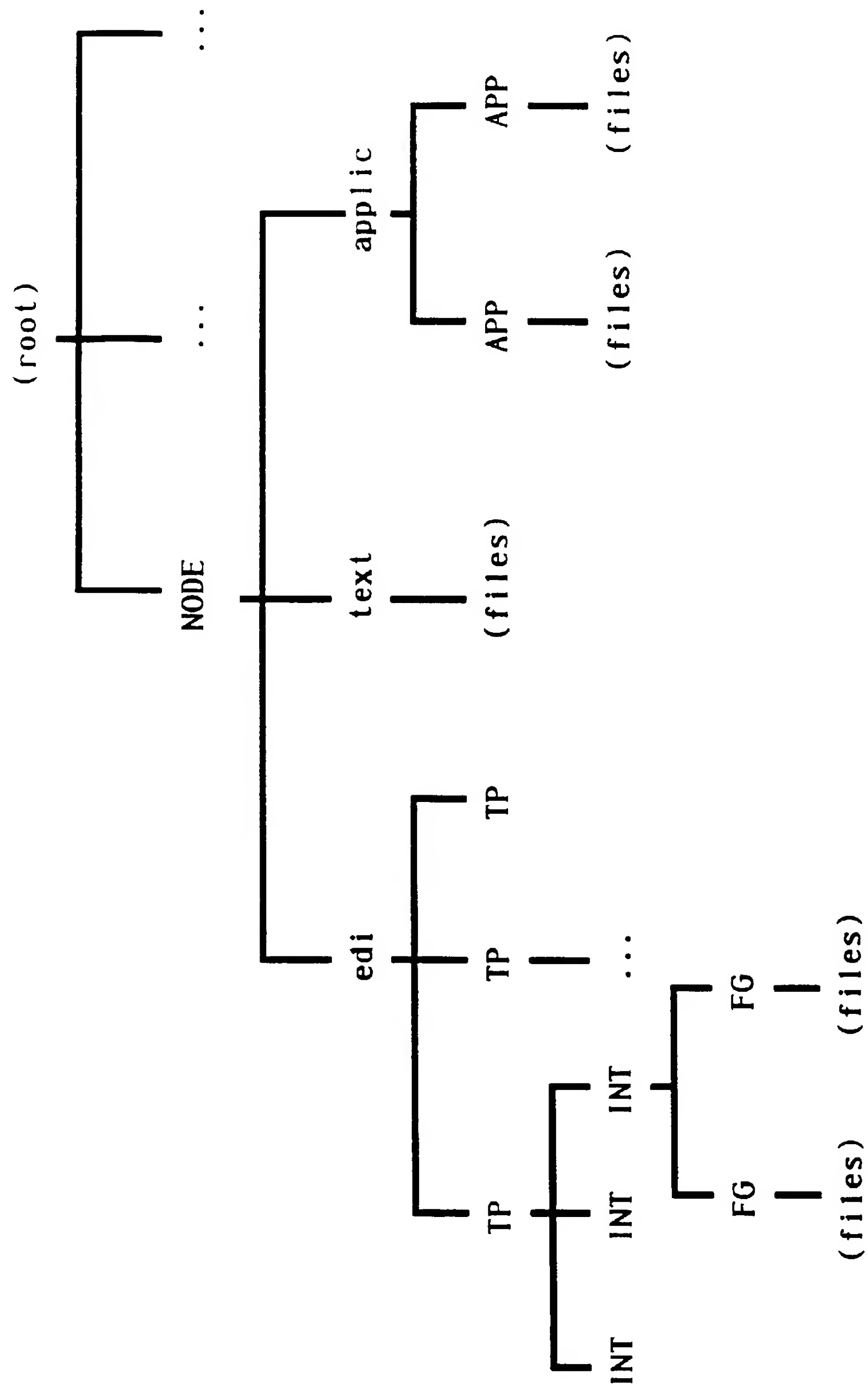
(root)
...
...
NODE
applic
APP
APP
(files)
(files)
text
(files)
edi
TP
TP
TP
...
INT
INT
INT
FG
FG
(files)
(files)

FIG. 6

```
                                    6110
AAAAA   CCCCC   SSSSS
A       C       S                  [MAIL]    Perry Lawrence                DUNS #

AAAAA   C       SSSSS   REMITTANCE   Bank of Trade                    99-999-9999
A    A  C           S       TO       9022 Lawrence Express Way
A    A  CCCCC   SSSSS                San Jose, CA 94520                     Page-
------------------------------------------------------------------------------
ORDER = CUSTOMER PO # DEPT SALESPRS  SHIP VIA               DATE        INVOICE =
004127                      150   100 UNITED PARCEL [GROUN] NOV 27 1989  101903-

CUSTOMER #          DIVISION                          6120|        TERMS
B1200-          JUNIORS DIVISION                               8% 10 EOM
------------------------------------------------------------------------------
   BILL ATHLETIC FASHIONS BILLING          SHIP BROADMORE
    TO  1445 JONES STREET                   TO 1400 WILSHIRE BLVD.
        SAN FRANCISCO       CA 94103            LOS ANGELES        CA 94800
******************************************************************************
STYLE     CLR  DIM  ***   ***   ***   ***   ***   DOZ/UN     PRICE      TOTAL
------------------------------------------------------------------------------
ASTYLE01 BLK         J     5     5     5     5     5 00        5         300
 (YR STYLE-123                  )
ASTYLE02 RED         G                               06        7          42
 (YR  STYLE-A00000000000001 )
  FREIGHT                                             1        2           2
                        HALF AIR FREIGHT
                        SHIP ORDER COMPLETE
******************************************************************************
TOTAL DOZ/UN-           5 06              MERCHANDISE TOTAL-             342
CARTONS-  1 WEIGHT-    5 LBS.               NON-MERCH TOTAL-               2
DUE DATE-      JAN 10 1990                    INVOICE TOTAL-             344
```

FIGURE 7

```
         7111        7112
BIG* [NOV271989] * [101903-]
N1*PE
N2*ATHLETIC FASHIONS BILLING
N3*1445 JONES STREET
N4 * SAN FRANCISCO*CA*94103
PER*AD*Perry Lawrence
N1*SD
N2*BROADMORE
N3*1400 WILSHIRE BLVD.
N4*LOS ANGELES*CA*94800
IT1*ASTYLE01*60*UN*5
CTP******SEL*300
PID*F****G
TDS*344*02*342
CTT*1
```

FIGURE 8

```
          defdocument doc810:
1100        area x12_heading:
     1101 seg ST: m 1
     1102 seg BIG: m 1
          seg NTE: f 100
          seg CUR: o 1
1110      loop loop_n1: 1 200
            seg N1: o 1
     1111   seg N2: 0 2
            seg N4: o 1
            seg REF: o 12
            seg PER: o 3
          endloop
          seg ITD: o 5
          seg DTM: o 1
          seg FOB: o 1
        endarea

1200    area x12_detail:
          loop loop_1: 1 99999
            seg ITI: m 1
            seg NTE: f 100
            seg CUR: o 1
            seg SLN: o 100
            seg J2X: o 10000
            seg IT3: o 1
            seg CTP: o 25
            seg IT4: o 1
            seg ITA: o 10
            seg ITD: o 2
            loop loop_n1_2: 1 200
               seg N1: o 1
               seg N2: o 2
               seg N3: o 3
               seg N4: o 1
               seg REF: o 12
               seg PER: o 3
              endloop
            seg DTM: o 10
            seg CAD: o 1
           endloop
         endarea

1300     area x12_summary:
            seg TDS: m 1
            seg CAD: o 1
            seg ITA: o 10
            seg ISS: o 2
            seg CTT: m 1
       1321 seg SE: m 1
         endarea
        enddef
```

FIGURE 9

```
2100 (defseg CUR 2105
        "CURRENCY"
        :used-in
             ( 810 830 832 840 843 850 855 856 860 861
                865 820 844 845 846 867 823 )

        :syntax
        ((-> (-or 8 9) 7)
         (-> (-or 11 12) 10)
         (-> (-or 14 15) 13)
         (-> (-or 17 18) 16)
         (-> (-or 20 21) 19))
        :elts
         ((M 0098)  (M 0100)  (C 0280)  (0 0098)  (0 0100) (0 0669 (C 0374)
     (0 373) (C 0337) (C 0374) (0 0373) (0 0337) (C 0374) (0 0373) (0 0337)
     (C 0374) (0 0373) (0 0337) (C 0374) (0 373) (0 0337) )
     :vers ( "X12-0.0")

2200    (defseg N1  2205

        2210 "NAME"
        2220 :used-in
            2225

          ( 810 830 832 840 843 850 855 856 860 861 865 104 105 107

            108 109 113 116 300 301 302 304 305 306 307 308 310 312 313 316
            317 422 423 424 425 820 875 876 877 880 881 882 884 885 888 889
            890 891 900 905 940 941 942 943 944 945 994 110 204 205 206 207
            208 210 214 216 309 874 878 879 920 892 893 824 844 845 846 862
            867 869 870 360 404 410 417 426 883 823 858 947 )
     2270 :syntax
          ((-or 2 (-and 3 4)))
          :elts
     2240 ((M 0098) (C 0093) (C 0066) (C 0067) )
     2250 :vers ( "X12-0.0"))
```

FIG. 10

```
3100
3110 ( defelt 0093 ("AN" 1 35 ) "NAME"
        ()             3113    3114
3120 :used-in (  ACT  G3  G61  N1  N2  PER  POD  SCH  S8)
3130   :vers (  "TDCC-1.5"))

3200
     ( defelt 0098  ("ID" 2  2)  "ENTITY IDENTIFIER CODE"
       ( "AC" "AD" "AG" "AK" "AO" "AS" "BK" "BL" "BN" "BO"
        "BS" "BT" "BW" "BY" "CA" "CB" "CC" "CL" "CM" "CN" "CO" "CP"
        "CR" "CS" "CT" "CV" "C1" "C2" "DB" "DC" "DE" "DS" "EC" "EE"
        "EM" "EP" "EX" "FH" "FR" "FW" "IA" "IC" "II" "IK" "IM" "IN"
3220    "IS" "IT" "LN" "LP" "MA" "MC" "MF" "MI" "MP" "NC" "NO" "NP"
        "NV" "N1" "N2" "OB" "OC" "OI" "OO" "OP" "OT" "OV" "OW" "PA"
        "PB" "PC" "PD" "PE" "PF" "PG" "PH" "PI" "PJ" "PM" "PN" "PP"
        "PR" "PS" "PT" "PU" "PV" "RB" "RC" "RD" "RE" "RH" "RL" "RM"
        "RO" "RP" "RR" "RS" "RT" "SA" "SC" "SD" "SE" "SF" "SG" "SH"
        "SI" "SM" "SN" "SO" "SP" "SS" "ST" "SU" "SW" "TN" "TO" "TR"
        "TS" "TT" "UC" "UP" "VN" "VO" "WH" "ZZ" "11"
      :used-in ( CUR D1 D5 F1 F5 M1 N1 PT PWK SCH U1 U5
                 Y1)
        :vers ( "TDCC-1.5"))
```

FIG. 11

```
4100 (defun x12>n verification-generator (descriptor)
       (set-edi>datatype-scaling-factor descriptor
                                 (- (aref (get-regular-expression-match "a")
                                           0 )
                                           ?0))
       (let*  ((maxlength  (get-edi-datatype-max-length descriptor))
               (minlength  (get-edi-datatype-min-length descriptor))
               (regx  (compile-regular-expression
                        (if (equal maxlength 1) "<0> [--] {1,0}[0-9] <'0>"
                          (concat "<0> [--] {1,0}(0[0-9]{"
                              (format "%d" (1-minlength))
                              "}|[1-9][0-9]{"
                              (format "%d, %d" (1-maxlength) (1-minlength))
                              "})<~0>")))))
            (set-edi-datatype-verification descriptor regx)
            (set-edi-datatype-max-length descriptor (1~ maxlength)))
     )

4200 (defun x12-r-verification-generator (descriptor)
       (set-edi-datatype-scaling-factor descriptor 0
       (let*  ((maxlength (get-edi-datatype-max-length descriptor))
               (minlength (get-edi-datatype-min-length descriptor))
               (regx (compile-regular-expression
                       (if (equal maxlength 1)
                           "<0>[-+]{1,0}(\\.[0-9]¦[0-9]|[0-9]\\.)<~0>"
                         (concat "<0>(0[0-9]+(<"
                            (format "%d" minlength)
                            ">|\\.[0-9]*<"
                            (format "%d" (1- minlength))
                            ">)|[1-9][0-9]*(<"
                            (format "%d-%d" minlength maxlength)
                            ">|\\.[0-9]*<"
                            (format "%d-%d" (1- minlength) (1- maxlength))
                            ">)|[+-]0[0-9]*(<"
```

FIG. 12A

```
                          (format "%d" (1-minlength))
                          ">|\\.[0-9]*<"
                          (format "%d" (- 2 minlength))
                          ">)|[+-][1-9][0-9]*(<"
                          (format "%d-%d" (1- minlength) (1- maxlength)
                          ">|\\.[0-9]*<"
                          (format "%d-%d" (- 2 minlength) (- 2 maxlength))
                          ">))<~0>")))))
  (set-edi-datatype-verification descriptor regx)
  (set-edi-datatype-max-length descriptor (- 2 maxlength)))
)
```

4300

```
;This version allows all blank values
(defun x12~an~verification-generator (descriptor)
 (let* ((maxlength (get-edi-datatype-max-length descriptor))
        (minlength (get-edi-datatype-min-length descriptor))

        (reqx (compile-regular-expression
               (if (equal minlength maxlength)
                   "<0>[]a-ij-rs-zA-IJ-RS-Z
0-9[!\"#&'()*+,./:;?=%$@_{}\\|<>-]*<'0>"
                    (concat "<0>[]a-ij-rs-zA-IJ-RS-Z
0-9[!\"#&'()*+,./:;?=%$@_{}\\|<>-]{"
                     (format "%d" minlength)
                     "}(<~0>|[]a-ij-rs-zA-IJ-RS-Z
0-9[!\"#&'()*+,./:;?=%$@_{}\\|<>-]*<'1>[]a-ij-rs-zA-IJ-RS-ZO
                                        -9[!\"#&'()*+,./:;
%$@_{}\\|<>-])<~0>"))
                   )))
    (set-edi-datatype-verification descriptor regx))
)
```

FIG. 12B

```
4400 (defdatatype ("X12-2.3")
      ("<0>[nN][0-9]Sa<~0>"  (x12-n-verification-generator))
      ("<0>[rR]<~0>" (x12-r-verification-generator)
      ("<0>[iI][dD]<~0>")
      ("<0>[aA][nN]<~0>" (x12-an-verification-generator))
      ("<0>[dD][tT]<~0>"
     ("<0>[0-9][0-9]((0(([13578](0[1-9]|[1-2][0-9]|30))|([469](0[1-9]|
       [1-2][0-9]|3
     ]))|(2(0[1-9]|[1-2][0-9]))))|(1(([02](0[1-9]|[12][0-9]|3[01]))|
       (1(0[1-9]|[12]
     9]|30)))))<~0>"))
       ("<0>[tT][mM]<~0>" ("<0>([01][0-9]|2[0-3])[0-5][0-9]<~0>"))
      )
```

FIG. 12C

```
'
' edi810_32.e
' Jessica Yang
'
' Data file: B_810.b
' Version  : 2.2
'
' Change: treetoedi --> write edi
' Also checks on mandatory and optional loops of edi standard
' with compliance check.
'
5100 function the_date(m,d,y)
     begin
         md=concat(m,d)
         mdy=concat(md,y)
         return mdy
     end

5200 filemap invoice
     begin
       loop
5210
       5212 record person: tree\HEADING\H_LOOP_N1[1]\PER[1][1]="AD"
                           tree\HEADING\H_LOOP_N1[1]\PER[1][2]=p_name
                           print "person is: ", p_name

            record header: inv_date=the_date(mon,day,year) 5313
                           tree\HEADING\BIG[1][1]=inv_date 5314
                           tree\HEADING\BIG[1][2]=inv_num 5315
                           print "invoice number is: ",inv_num

            record names:  tree\HEADING\H_LOOP_N1[1]\N1[1][1]="PE"
                           tree\HEADING\H_LOOP_N1[2]\N1[1][1]="SD"
                           tree\HEADING\H_LOOP_N1[1]\N2[1][1]=b_name
                           tree\HEADING\H_LOOP_N1[2]\N2[1][1]=s_name
```

FIG 13A

```
                  print "bill to: ",b_name
                  print "ship to: ",s_name

record addrs:     tree\HEADING\H_LOOP_N1[1]\N3[1][1]=b_addr
                  tree\HEADING\H_LOOP_N1[2]\N3[1][1]=s_addr
                  print "bill to addr: ",b_addr
                  print "ship to addr: ",s_addr

record locat:     tree\HEADING\H_LOOP_N1[1]\N4[1][1]=city
                  tree\HEADING\H_LOOP_N1[1]\N4[1][2]=state
                  tree\HEADING\H_LOOP_N1[1]\N4[1][3]=zip
                  tree\HEADING\H_LOOP_N1[2]\N4[1][1]=city1
                  tree\HEADING\H_LOOP_N1[2]\N4[1][2]=state1
                  tree\HEADING\H_LOOP_N1[2]\N4[1][3]=zip1

record unused: n=1
loop
record items:     if dozens > 0 then
                     if unit > 0 then
                         num_of = dozens * 12 - unit
                  else
                         num_of = dozens * 12
5211              else
                     num_of = unit

                  total=num_of * price

                  tree\DETAIL\D_LOOP_IT1[n]\IT1[1][1]=style
                  tree\DETAIL\D_LOOP_IT1[n]\IT1[1][2]=num_of
                  tree\DETAIL\D_LOOP_IT1[n]\IT1[1][3]="UN"
                  tree\DETAIL\D_LOOP_IT1[n]\IT1[1][4]=price
                  tree\DETAIL\D_LOOP_IT1[n]\CTP[1][6]="SEL"
                  tree\DETAIL\D_LOOP_IT1[n]\CTP[1][7]=total

          tree\DETAIL\D_LOOP_IT1[n]\D_LOOP_SLN[1]\PID[1][1]="F"
          tree\DETAIL\D_LOOP_IT1[n]\D_LOOP_SLN[1]\PID[1][5]=dimm
```

FIG 13B

```
                                  print "
                                  print "style is: ",style
                                  print "color is: ",color
                                  print "dimm is:  ",dimm
        5211                      print "num_of is: ",num_of
                                  print "price is: ",price
  5210                            print "total is" ",total
                                  n=n+1
          endloop

          record summy: inv_total = mer_total - non_total
                        tree\SUMMARY\TDS[1][1]=inv_total
                        tree\SUMMARY\TDS[1][2]=non_total
                        tree\SUMMARY\TDS[1][3]=mer_total
                        print "invoice total is: ",inv_total

                        tree\SUMMARY\CTT[1][1]=1
        endlood
5200  end

  5300     record person
           begin
              filler pic x(*)                        5311
              filler pic x(4)   value is "MAIL"      5312
     5310     filler pic b(*)
              p_name pic x(14)                       5313
              filler pic x(*)                        5314
              filler pic x(5)   value is "GROUN"     5315
           end

           record header
```

FIG. 13C

5300

```
begin
   filler pic b
   mon pic x(3) 5321
   filler pic b
   day pic x(2) 5323
   filler pic b
   year pic x(4) 5325
   filler pic b(*)
   inv_num pic x(*) 5327
   filler pic x                value is "\n"
end

record names
begin
   filler pic x(*)
   filler pic x(5)  value is "BILL "
   b_name pic x(25)
   filler pic b(*)
   filler pic x(5)  value is "SHIP "
   s_name pic x(9)
end

record addrs
begin
   filler pic x(*)
   filler pic x(3)   value is "TO  "
   b_addr pic x(25)
   filler pic b(*)
   filler pic x(3)  value is  "TO  "
   s_addr pic x(*)
   filler pic x     value is  "\n"
end
```

FIG. 13D

```
record locat
begin
   filler pic b(*)
   city   pic x(13)
   filler pic b(*)
   state  pic x(2)
   filler pic b(*)
   zip    pic x(5)
   filler pic b(*)
   city1  pic x(11)
   filler pic b(*)
   state1 pic x(2)
   filler pic b(*)
   zip1   pic x(5)
   filler pic x(*)
   filler pic x            value is "\n"
end

record unused
begin
   filler pic x(*)
   filler pic x(2)         value is "-\n"

record items
begin
   filler pic b(*)
   style pic x(8)
   filler pic b(*)
   color pic x(3)
   filler pic b(*)
   dimm pic x(1)
5300   filler pic x(35)
   dozens pic z
   filler pic b
   unit   pic zz
```

FIG. 13E

```
         filler pic b(*)
         price  pic z
         filler pic x(*)
         filler pic x(2)        value is ")\n"
      end

      record summy
      begin
         filler pic x(*)
         filler pic x(18)       value is "MERCHANDISE TOTAL-"
         filler pic b(*)
         mer_total pic zzz
         filler pic x(*)
         filler pic x(16)       value is "NON-MERCH TOTAL-"
         filler pic b(*)
         non_total pic z
         filler pic x(*)
         filler pic x           value is "\n"
      end

      begin
     5410 tree=makedoc("810", "002002")
5400 5420 mapfile(stdin,invoice)
     5430 write edi tree element="*", segment="\n", subelement="@"
      end
```

FIG. 13F

```
13110   record header
        begin
            ehcusn pic x(5)
            ehedpo pic x(15)
            ehtrid pic x(9)
            ehedlk pic x(9)
            ehdppo pic x(10)
            ehname pic x(30)
            ehadd1 pic x(30)
            ehadd2 pic x(30)
            ehcity pic x(20)
            ehstcd pic x(2)
            ehzip  pic x(9)
            ehcnty pic x(16)
            ehedct pic x(15)
            ehedph pic x(15)
            ehcarn pic x(15)
            ehcust pic x(5)
            ehtord pic x(3)
            ehedpf pic x(2)
            eheddt pic x(4)
            ehedtm pic x(3)
            eherfd pic x(70)
            ehdcmd pic x(1)
            ehorc1 pic x(4)
            ehlmdt pic x(4)
            ehtyme pic x(4)
        end

        record note
        begin
            eicusn pic x(5)
            eiedpo pic x(15)
            eitrid pic x(9)
            eintqf pic x(3)
            eispis pic x(2)
```

FIG. 14A

```
        eispin pic x(50)
        eiorc1 pic x(4)
        eilmdt pic x(4)
        eityme pic x(4)
end

record detail
begin
        edcusn pic x(5)
        ededpo pic x(15)
        edtrid pic x(9)
        edorc1 pic x(4)
        edlmdt pic x(4)
        edtyme pic x(4)
end

record detail1
begin
        elcusn pic x(5)
        eledpo pic x(15)
        eltrid pic x(9)
        ellisq pic x(2)
        elitnb pic x(15)
        elorqt pic x(4)
        elorqu pic x(2)
        eledpr pic x(5
        eledpm pic x(2)
        elsprf pic x(10)
        elitds pic x(25)
        elrqdt pic x(4)
        elcspt pic x(15)
        elcarn pic x(15)
        elcsri pic x(3)
        elitcg pic x(2)
        eldivn pic x(4)
        elorc1 pic x(6)
```

FIG. 14B

```
        elprcd pic x(4)
        ellmdt pic x(4)
        eltyme pic x(4)

record detsln
begin
        escusn pic x(5)
        esedpo pic x(15)
        estrid pic x(9)
        eslisq pic x(2)
        esstsq pic x(2)
        esstqt pic x(4)
        esstqu pic x(2)
        esqf01 pic x(2)
        esfd01 pic x(30)
        esqf02 pic x(2)
        esfd02 pic x(30)
        esqf03 pic x(4)
        esfd03 pic x(4)
        esorc1 pic x(4)
        eslmdt pic x(4)
        estyme pic x(4)
end

record detmea
        emcusn pic x(5)
        emedpo pic x(15)
        emtrid pic x(9)
        emlisq pic x(2)
        emmsrf pic x(2)
        emdmqf pic x(3)
        emdimu pic x(2)
        emdimd pic x(6)
        emdimf pic x(5)
        emstsq pic x(2)
        emspis pic x(4)
```

FIG. 14C

```
        emorc1 pic x(4)
        emlmdt pic x(4)
        emtyme pic x(4)
    end

    record detnte
    begin
        eccusn pic x(5)
        ecedpo pic x(15)
        ectrid pic x(9)
        eclisq pic x(2)
        ecspis pic x(2)
        ecntqf pic x(3)
        ecspin pic x(50)
        ecorc1 pic x(4)
        eclmdt pic x(4)
        ectyme pic x(4)
    end

    begin
13200   read edi podoc

    'Building first header record
        HEAD =PODOC\HEADINGd
13310           ehedpo=head\beg[1][3]
                ehdppo=head\ref[1][2]
                ehedct=head\per[1][2]
                ehedph=head\per[1][4]
                ehcarn=head\td5[1][5]
        NAM=PODOC\HEADING\N1LOOP[1]
                ehname=NAM\N1[1][2]
                ehcusn=NAM\N1[1][4]
                ehadd1=NAM\n3[1][1]
                ehadd2=NAM\n3[1][2]
                ehcity=NAM\n4[1][1]
                ehstcd=NAM\n4[1][2]
                ehzip=NAM\n4[1][3]
                ehcnty=NAM\n4[1][4]
```

FIG. 14D

```
             ehcust=podoc\heading\n1loop[2]\n1[1][4]
13400        putrec header

        'Building second header for note segment
             eiedpo=head\beg[1][3]
             eicusn=head\n1loop[1]\n1[1][4]
        putrec note

             'Building first detail record
             FOREACH P1 IN PODOC\DETAIL\D_LOOP DO
             begin
                ededpo=head\beg[1][3]
                edcusn=head\n1loop[1]\n1[1][4]
                'Building second detail record
                eledpo=head\beg[1][3]
                elcusn=head\n1loop[1]\n1[1][4]
                p1=podoc\detail\d_loop[n]
                ellisq=p1\po1[1][1]
                elorqt=p1\po1[1][2]
                elorqu=p1\po1[1][3]
                elitnb=p1\po1[1][7]
                elitds=p1\po1[1][9]
                elcspt=p1\po1[1][11]
                eledpr=p1\ctp[1][3]
                eledpm=p1\ctp[1][5]
                elsprf=p1\ref[1][2]
                elrqdt=p1\dtm[1][2]
                elcarn=p1\td5[1][5]

                'Building third detail record, mea segment
                FOR EACH S1 IN P1\SLN DO
                begin
                   esedpo=podoc\heading\beg[1][3]

             escusn=podoc\heading\n1loop[1]\n1[1][4]
                   eslisq=S1[1]
                   esstsq=S1[2]
```

FIG. 14E

```
        esstqt=S1[4]
        esstqu=S1[5]
        esqf01=S1[9]
        esfd01=S1[10]
        esqf02=S1[11]
        esfd02=S1[12]
        esqf03=S1[13]
        esfd03=S1[14]
    end

    'Building fourth detail record
    FOR EACH M1 IN P1\MEA DO
    begin
        emedpo=podoc\heading\beg[1][3]

emcusn=podoc\heading\n1loop[1]\n1[1][4]
        emlisq=p1\po1[1][1]
        emmsrf=M1[1]
        emdmqf=M1[2]
        emdimu=M1[4]
        emdimd=M1[5]
        emdimf=M1[6]
    end

    'Building fifth detail record
    begin
    ecedpo=podoc\heading\beg[1][3]
    eccusn=podoc\heading\n1loop[1]\n1[1][4]
    eclisq=p1\po1[1][1]
    end
end
```

FIG. 14F